Sistemas Informáticos
Curso 2005 - 2006

Estudio de una jerarquía de memoria controlada por software

Carlos Beltrán Segovia
Elisa Gil González-Madroño
Manuel Martínez Herranz

Dirigida por:

Prof. José Ignacio Gómez Pérez

Dpto. Arquitectura de Computadores y Automática

Los autores de este proyecto de sistemas informáticos autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a los autores, tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

Elisa Gil Manuel Martínez Carlos Segovia

Madrid 26 de Septiembre 2006

ÍNDICE

1. RESUMEN DEL PROYECTO

Los sistemas empotrados representan un gran porcentaje de las ventas en el mercado de dispositivos electrónicos. Cada vez con mayor asiduidad, dichos sistemas incorporan procesadores de propósito general (ARM6, ARM7….).Sin embargo, las aplicaciones incluidas en los sistemas modernos , son cada vez más exigentes.

En la actualidad, el mercado de sistemas empotrados está en gran auge, y es por ello que factores como la energía, potencia, consumo son de gran importancia en multimedia y dispositivos móviles o wireless.

Para conseguir un rendimiento adecuado, manteniendo un consumo de potencia bajo, se ha propuesto el uso de memorias estáticas controladas por software (scratch-pad), en lugar de las tradicionales memorias caché controladas por hardware.

Se busca que las fuentes de estos dispositivos duren lo máximo posible, pudiendo proporcionar los picos de energía suficientes para el correcto funcionamiento del dispositivo. Es por ello, que el consumo juega un papel fundamental en los sistemas empotrados para preservar lo máximo las fuentes de energía.

El presente proyecto pretende explorar las posibilidades de dichas jerarquías de memoria centrándose en un conjunto de algoritmos significativos, en los cuales se calculará el consumo de energía asociado a su ejecución, tanto para jerarquías de memoria con caché, con caché y scratch-pad, y con solo scratch-pad. Así mismo también se calculará el número de accesos a cada elemento de memoria y el número de ciclos de ejecución.

Para ello, se utilizará el simulador arquitectónico SimpleScalar, incorporando memorias SRAM (scratch-pad) y un controlador DMA, bajo un procesador ARM 7.

Se realizarán una serie de pruebas sobre algoritmos significativos y mediante la adaptación del modelo de consumo de energía del simulador arquitectónico SimpleScalar, analizaremos los resultados.

2. INTRODUCCION

El interés por los sistemas que hoy en día se denominan empotrados (embedded), ha sufrido un aumento considerable en la última década con su aplicación a los sistemas denominados de tiempo real. No obstante, se puede encontrar referencias a este tipo de sistemas que datan de la época de los 70.

Antes de entrar en materia, se debería dar una definición de los sistemas que se van a tratar. En primer lugar es preciso indicar que no existe ninguna definición estándar de lo que se entiende por sistema empotrado; es más, dependiendo del área en el que se encuentre las respuestas son diferentes. Así se puede encontrar con las siguientes definiciones:

- Un sistema empotrado es un sistema operativo ejecutándose en un micro de pocos recursos.
- Un sistema empotrado es un artefacto (hardware + software) no susceptible de modificación del algoritmo que define su comportamiento.
- Un sistema empotrado es un procesador, con sus elementos externos que desarrolla una función específica de manera autónoma.
- Un sistema empotrado es un sistema computador de propósito especial construido en un dispositivo mayor.
- Un sistema empotrado es una mezcla de hardware y software que constituye un componente dentro de un sistema mas complejo y se espera que funcione sin intervención humana.

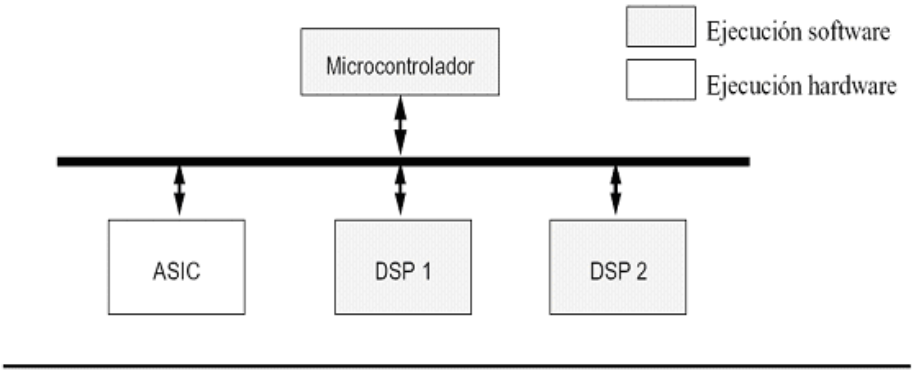

Figura 1: mezcla de ejecución hardware software en un sistema empotrado

En la [Figura1], observamos esa mezcla de ejecución software y ejecución hardware dentro de un sistema. Tenemos una plataforma ASIC y dos DSP (procesadores digitales de señales).

En la actualidad, el mercado de sistemas empotrados está en gran auge, y es por ello que factores como la energía, potencia, consumo son de gran importancia en multimedia y dispositivos móviles o wireless.

Se busca que las fuentes de estos dispositivos duren lo máximo posible, pudiendo proporcionar los picos de energía suficientes para el correcto funcionamiento del dispositivo. Es por ello, que el consumo juega un papel fundamental en los sistemas empotrados para preservar lo máximo las fuentes de energía.

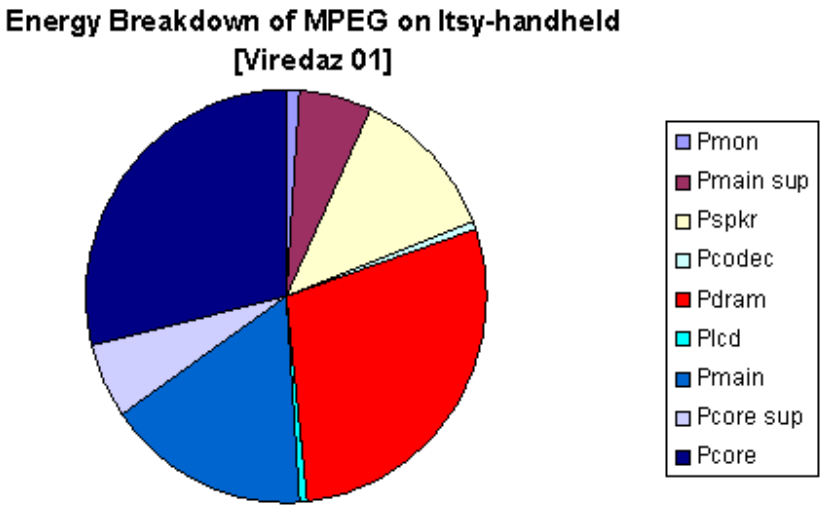

Figura 2: principales fuentes de consumo en un MPEG

En la [Figura2], tenemos como una de las principales fuentes de consumo de energía reside en la memoria. En el ejemplo de la figura observamos la energía que se consume en un MPEG, donde la SDRAM y la SRAM son la principales fuentes de consumo. Una reducción de ese consumo de energía, mediante el estudio de diferentes configuraciones o jerarquías de memoria, sería muy interesante para aumentar la duración de las fuentes de energía.

En la [Figura 3], tenemos los sistemas empotrados que se pueden encontrar en un hogar medio clasificados de forma general. Se distinguen cuatro grandes grupos de elementos: los elementos basados en PC (como pueden ser el ordenador, un PDA o cualquier elemento de entrada/salida como una impresora); los elementos relativos a las comunicaciones (como pueden ser los teléfonos o intercomunicadores); los elementos relativos al entretenimiento (como pueden ser los televisores, videos y DVD); y los elementos relativos al confort y comodidad (como pueden ser los electrodomésticos y climatizadores).

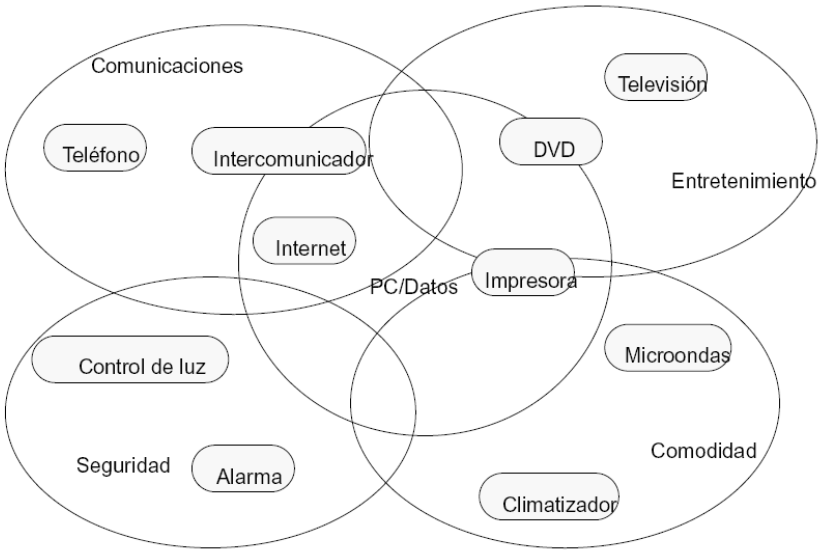

Figura 3: sistemas empotrados que se encuentran en un hogar medio

En la [Figura 4] tenemos un ejemplo del interior de un sistema empotrado. Se trata de un router, en donde las etiquetas caracterizan las principales regiones. Destacamos (4) Microprocesador (6) RAM y (7) Flash Memory.

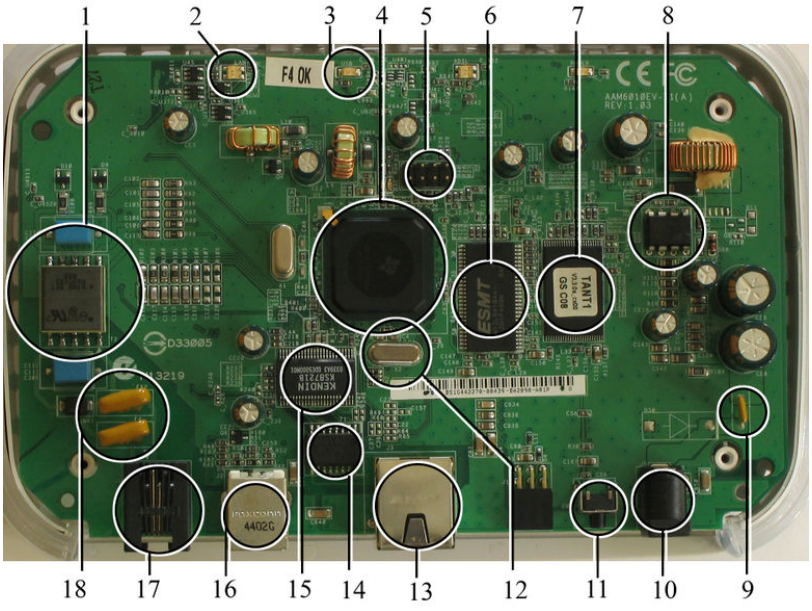

Figura 4: interior de un sistema empotrado.

En la [Figura 5], se muestra el primer sistema empotrado moderno reconocido, “The Apollo Guidance Computer”.

Figura 5: primer sistema empotrado

3. ARQUITECTURA SIMULADA

Pasamos a describir la arquitectura simulada, que consta de un microprocesador ARM y una jerarquía de memoria compuesta por una memoria caché, una memoria scratch-pad y un sistema de acceso directo a memoria DMA.

3.1 ARM

Se denomina ARM a una familia de microprocesadores RISC diseñados por la empresa Acorn Computers y desarrollados por Advanced RISC Machines Ltd., una empresa derivada de la anterior. Su característica principal es el bajo consumo de energía y su bajo costo.

En el mercado actual de los sistemas empotrados, son los grandes dominadores. Esto queda reflejado en los siguientes gráficos.

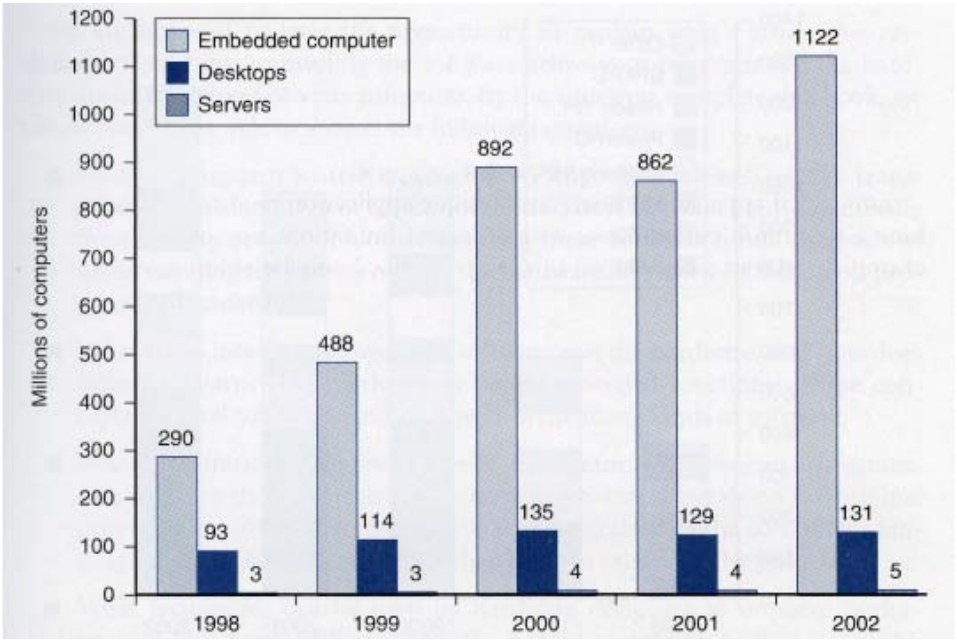

Figura 6: situación del mercado actual

En la [Figura 6], vemos la venta de los distintos tipos de procesadores entre los años 1998 y 2002, con gran impacto en los sistemas empotrados con un gran aumento.

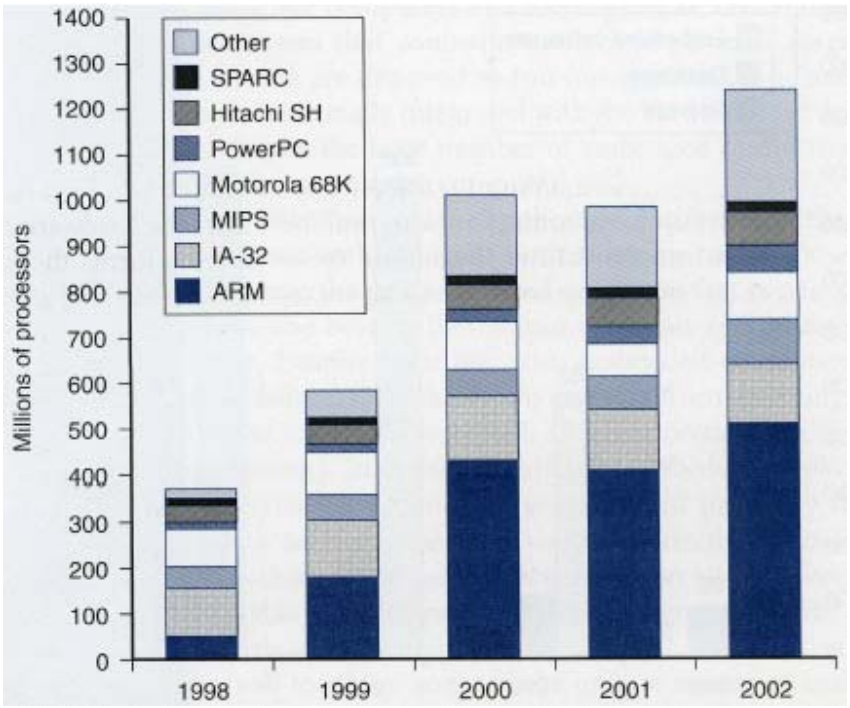

Figura 7: ventas de los distintos procesadores entre los años 98-02

En la [Figura 7], vemos la venta de procesadores RISC entre los años 1998 y 2002, en donde destacan los procesadores ARM.

Existen varias familias de procesadores ARM (ARM7,ARM9,ARM10 y ARM11). Dentro de la familia de ARM 7, los procesadores utilizados principalmente son ARM7TDMI, ARM7TDMI-S, ARM7EJ-S, ARM720T. Fue creado para tener un mejor desempeño dentro de las limitaciones de ser simples, ocupar poco área, y tener un bajo consumo de energía. Para ocupar poco área, deja a los coprocesadores tareas secundarias como I/O, operaciones de punto flotante, etc. Para el bajo consumo de energía, simplifica los circuitos.Tiene un pipeline corto (operando a bajas frecuencias) y acciones que hacen que si el procesador no estuviera activo, se consuma poco.

En la arquitectura destaca que son microprocesadores de propósito general de 32 bits, con una arquitectura RISC, una estructura Von Neumann con un rendimiento de hasta 120 MIPS y un bajo consumo de energía (80mW).

En la [Figura 8], motramos la evolución de pipeline, desde uno de tres etapas al utilizado de cinco etapas. Inicialmente se tenían las etapas (Fetch, Decode, Execute) y ahora tenemos (Fetch, Decode, Execute, Memory y Write). El ancho de banda es de una instrucción por ciclo.

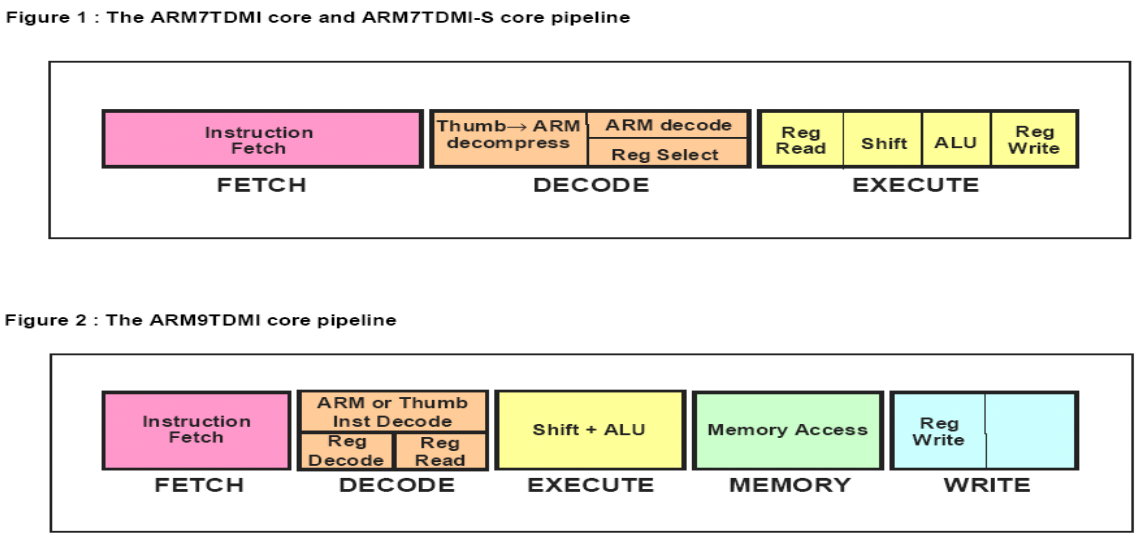

Figura 8: evolución del pipeline

En la [Figura 9], mostramos el esquema interno de un procesador ARM 7, con un banco de registros, con dos puertos de lectura y uno de escritura de acceso a todos los registros y un puerto de lectura y otro de escritura adicionales para acceder a PC. Un desplazador que desplaza el contenido de los registros (barrel shifter). Un registro que almacena las direcciones de acceso a memoria (address register + incrementer). Un registro de información que almacena los datos en las transferencias con memoria (data register) y un descodificador de instrucciones y control (instruction decode & control).

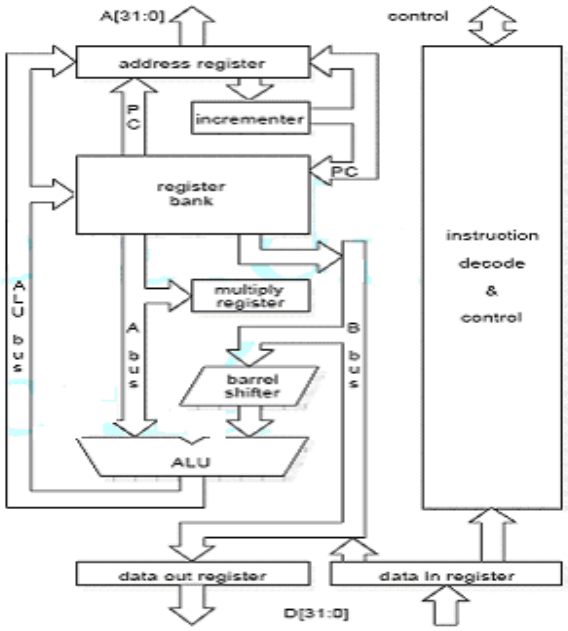

Figura 9: esquema interno de un procesador ARM7

La tecnología ARM está licenciada en varios dispositivos de compañías como: Philips, Atmel, Frescale, Cirrus, Hyundai, Intel, Oki, Samsung, HP, IBM, Sony. Estos dispositivos se pueden englobar en tres grandes grupos como muestra la [Figura 10].

Figura 10: englobamiento de dispositivos ARM.

Y un ejemplo en particular lo mostramos en la [Figura 11] siguiente, en donde tenemos una plataforma ASIC, en donde tenemos un procesador ARM Macrocell con sus conexiones a la lógica empotrada.

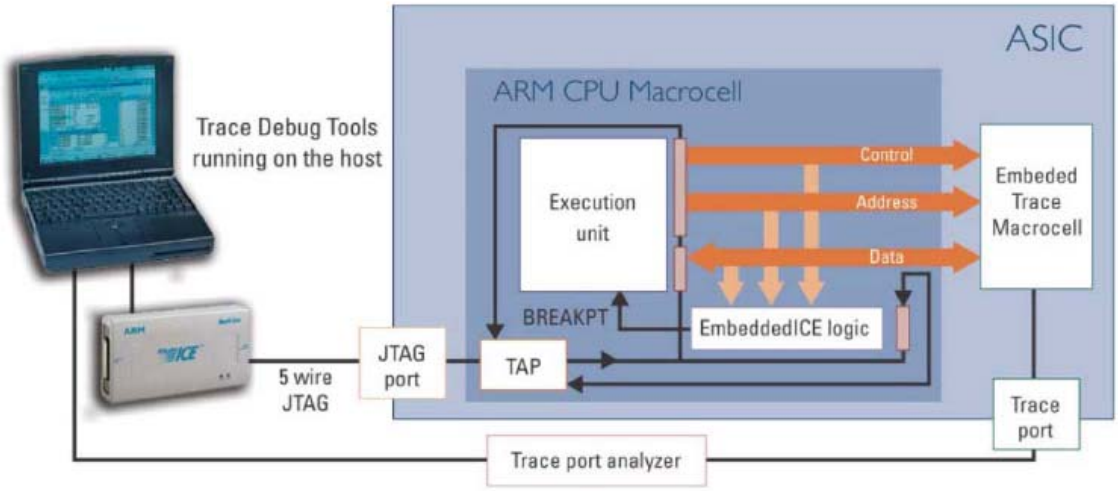

Figura 11: plataforma ASIC.

3.2 Jerarquía de memoria

3.2.1 Memoria caché

En la [Figura 12], vemos la organización de una memoria caché, en donde sus componentes principales son un decodificador, un array de etiquetas, una columna de multiplexores, amplificadores, comparadores de etiquetas, drivers de salida para etiquetas, un array de datos, una columna de multiplexores de datos, amplificadores para los datos y drivers de salida para los datos. Se observan dos bloques diferenciados para las etiquetas y para los datos. La energía total consumida por la caché es la suma de la energía que consumen todos los componentes que la integran. El comportamiento a priori es difícil de predecir.

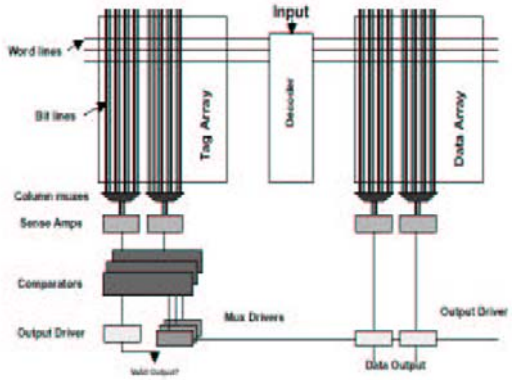

Figura 12: organización de una memoria caché

En la [Figura 13], mediante un diagrama de flujo , representamos su funcionamiento. Una vez que se recibe la dirección física de la CPU, se busca si el contenido de la dirección se encuentra en un bloque de la memoria caché (acierto). Si no existe (fallo) se busca el bloque en memoria principal y se asigna un marco de la memoria caché al bloque de la memoria principal.

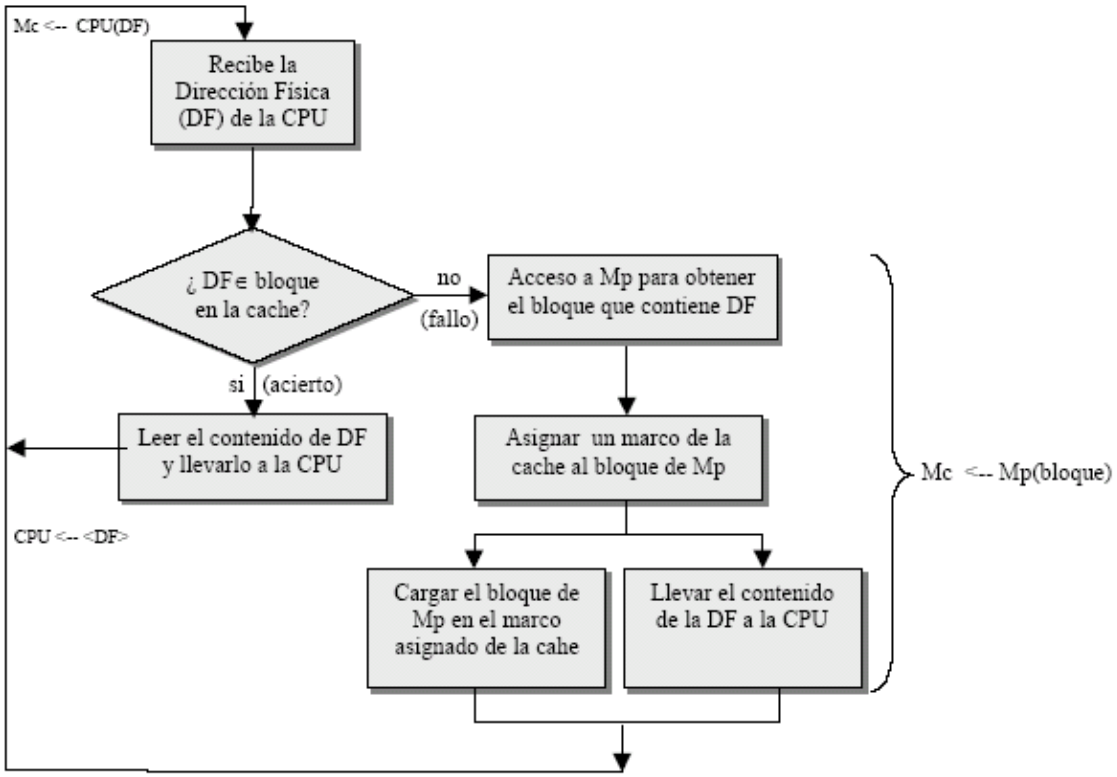

Figura 13: funcionamiento de la caché

3.1.1. Memoria scratch-pad

En la [Figura 14] y [Figura 15], tenemos la disposición de una memoria scratch-pad. Esta memoria tiene un array de datos con una parte de decodificación y circuitos lógicos. Este modelo esta diseñado para introducir en la scratch-pad los objetos en la ultima fase del compilador o directamente por el programador. No se necesita chequear la disponibilidad de la instrucción/datos en la scratch-pad. Esto hace reducir el comparador anterior y las señales de acierto/fallo de la caché. Con ello se reduce energía y área. El array de datos esta formado por celdas. Se observa que cada celda esta formada por seis transistores.

El área es otro de los puntos importantes que tiene primordial importancia para la jerarquía de memoria. Al no tener bloque de etiquetas en la scratch-pad, se ahorra en ello.

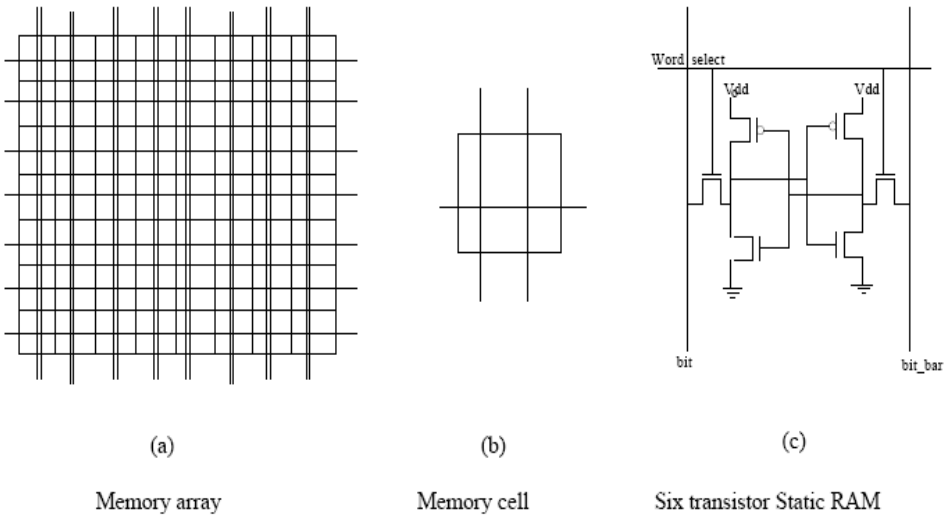

Figura 14: disposición de una memoria scratch

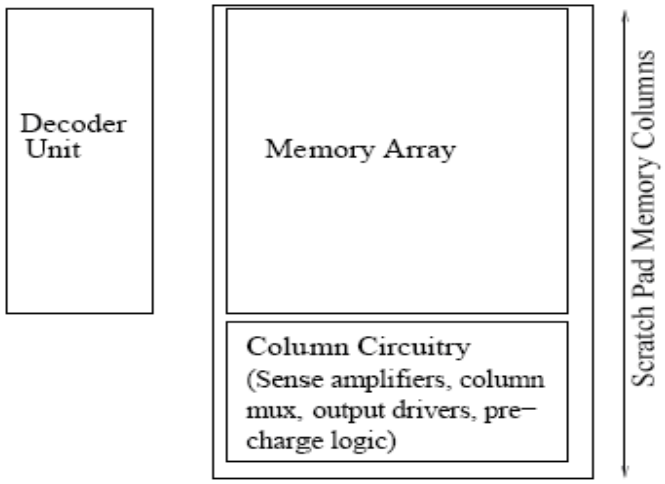

Figura 15: disposición de una memoria scratch

La energía consumida por la Scratch-Pad es una estimación de la energía gastada por los circuitos de decodificación y las columnas de memoria.

La energía en el array de memoria consiste en la energía consumida en sus amplificadores, multiplexores, los circuitos de salida y las celdas de memoria. La mayor parte de energía que se gasta parte del array de memoria.

Al ser fijado un espacio fijo de direcciones donde reside la scratch-pad, al traducir la dirección física generada por la CPU, sabremos en todo el momento si estamos en la memoria scratch-pad, con lo que el comportamiento no es difícil de predecir.

3.2.3 DMA

Las transferencias de datos a memoria caché son gestionadas por su controlador. Éste se encarga de ir a memoria principal a por el dato cada vez que es necesario. Son transparentes a la CPU, que puede seguir realizando otras tareas. Sin embargo, las transferencias que son necesarias para este proyecto, las transferencias entre scratch y memoria principal, no se pueden llevar a cabo de esa manera. Hay que transferir los datos de modo manual porque no se dispone de un módulo que se encargue de ellas.

Una posibilidad es hacerlo directamente en el código fuente de la aplicación a simular, es decir encargar al procesador de las transferencias. Eso supondría que la CPU dejase de realizar otras tareas, ralentizando el tiempo de ejecución de los programas.

Otra posibilidad, la más interesante desde el punto de vista de este proyecto, es utilizar un DMA que se encargue de transferir datos liberando al procesador de esa tarea.

Habitualmente un DMA necesita tres parámetros fundamentales: la dirección del origen, la dirección del destino y el número de palabras o bytes a transferir. Normalmente la dirección origen suele ser un periférico pero en este proyecto la dirección origen siempre será una dirección de la memoria principal. Es decir que la función del controlador de DMA es básicamente escribir en tres registros de control.

En la Figura se puede ver cómo implementar un DMA en una arquitectura como la utilizada en este proyecto, es decir con memoria scratch. Se puede apreciar que el DMA conecta la memoria scratch con el procesador. El DMA consta del controlador y de la unidad de transferencias. El controlador es la interfaz entre el procesador y el DMA y es el encargado de iniciar una transferencia. Para ello, cuando recibe una petición de transferencia por parte del procesador consulta el estado de la unidad de transferencias y si está libre inicia la transferencia.

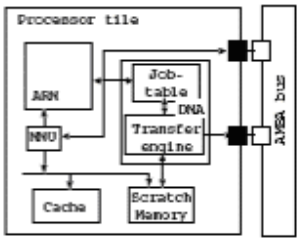

Figura 16: DMA en una arquitectura con memoria scratch.

En este proyecto, debido a las aplicaciones que hacen uso de él, principalmente multimedia, también se requería el uso de transferencias de estructuras bidimensionales como pueden ser matrices. Para este tipo de transferencias se necesitan más parámetros que los tres ya comentados previamente. Los más inmediatos son la dirección origen representada por un aspa en la Figura, la dirección destino, el número de filas y el número de columnas (o bytes de cada fila) que nos queremos llevar de la matriz origen. Otros de los menos intuitivos son el tamaño de la matriz grande (o número de bytes que ocupa una fila) necesario para poder hallar el elemento coloreado de rojo ya que se halla sumándolo a la dirección origen y el número de columnas de la dirección destino.

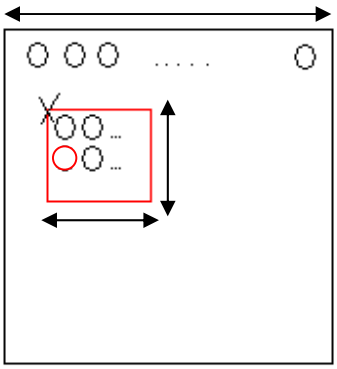

Figura 17: parámetros necesarios para una transferencia por bloques.

En la siguiente figura mostramos parte del código de la multiplicación de matrices por bloques. No se muestra todo el código porque es bastante largo y bastan estas líneas para mostrar un ejemplo de transferencia por bloques.

```
int main(){

  int i0,j0,k0,i,j,k;
  int idX, idY, idZ;

  int sum1;
  int *X_act, *X_next, *X_aux, *Y_act, *Y_next, *Y_aux, *Z_act, *Z_next, *Z_aux;
  int offset = 4*dim_bloque*dim_bloque;

  init();

  X_act = (int *)SCRATCH1;
  X_next = (int *)&SCRATCH1[offset];
  Y_act = (int *)&SCRATCH1[offset*2];
  Y_next = (int *)&SCRATCH1[offset*3];
  Z_act =(int *)&SCRATCH1[offset*4];
  Z_next =(int *)&SCRATCH1[offset*5];

  (1) idX=syscall(219,X,N*4,X_act,dim_bloque*4,dim_bloque,dim_bloque*4);
  (2) idY=syscall(219,Y,N*4,Y_act,dim_bloque*4,dim_bloque,dim_bloque*4);
  (3) idZ=syscall(219,Z,N*4,Z_act,dim_bloque*4,dim_bloque,dim_bloque*4);

  for (i0=0 ; i0<N ; i0+=dim_bloque){
```

Figura 18: ejemplo de una transferencia por bloques

Las líneas numeradas con 1,2 y 3 representan una transferencia por bloques. Comentaremos por ejemplo la transferencia de la X precedida por el número 1, El primer parámetro es el código de la syscall, el segundo representa la dirección origen, en este caso la primera posición de la X. Si quisiéramos hacer la transferencia desde el elemento diez de la X pondríamos &X[10]. Es importante el operador & ya que indica dirección de.

El tercer parámetro es el número de bytes que ocupa una fila de la matriz origen que ya comentamos previamente para qué era necesario. El cuarto parámetro nos indica la dirección destino, en este ejemplo es un array auxiliar situado en la scratch. El quinto nos indica el número de bytes que tiene la matriz destino. Este parámetro siempre será igual al número de bytes de cada fila que queramos copiar. El sexto representa el número de filas que nos queremos llevar de la matriz origen y el último representa cúantos bytes queremos llevarnos de cada fila de la matriz origen. En la figura podemos ver la transferencia que estamos realizando en realidad.

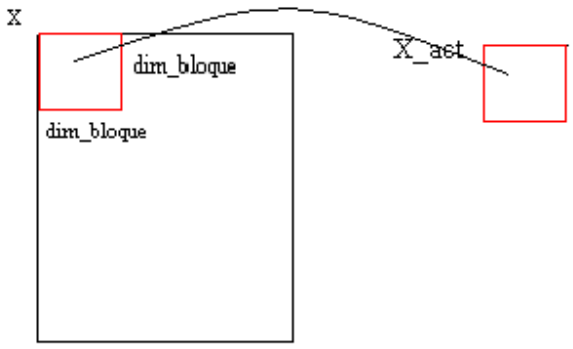

Figura 19: transferencia real

4. TRABAJO DESARROLADO.

4.1 SimpleScalar: un simulador arquitectónico.

SimpleScalar se escribió en 1992 como parte de un proyecto en la universidad de Wisconsin bajo la dirección de Gurindar S. Sohi. En 1995, con la ayuda de Doug Burger, el programa pasó a ser de libre distribución disponible para uso académico. La web encargada de su distribución es http://www. simplescalar.com.

Desde su aparición SimpleScalar ha llegado a ser muy popular entre los investigadores y profesores de arquitectura de computadores. Por ejemplo, en 2000 más de un tercio de las mejores publicaciones de arquitectura usaban SimpleScalar para evaluar los diseños.

El objetivo de SimpleScalar es acelerar el desarrollo del hardware, empleando modelos software de lo que se desea construir. Éstos se implementan en lenguajes de programación tradicionales o en lenguajes de diseño de hardware y luego se simulan con una carga de trabajo apropiada. Luego se ejecutan para medir el rendimiento y la corrección de su diseño. Estos modelos se deben utilizar para desarrollar y probar distintas alternativas de diseño antes de llegar a construir el circuito real. Además pueden construirse más rápidamente que su equivalente hardware.

Hay tres parámetros críticos que marcan una buena implementación del modelo software que son: el rendimiento, la mantenibilidad y el nivel de detalle. El rendimiento determina la carga real de trabajo que puede simular. La mantenibilidad permite hacer modificaciones sin mucho trabajo permitiendo hacer pequeñas variaciones en los modelos sin demasiado coste adicional. El nivel de detalle define el grado de abstracción utilizado para implementar los componentes del modelo.

En la práctica, optimizar estas tres características a la vez es difícil. De hecho, la mayoría de las implementaciones optimizan una o dos de las tres lo que explica por qué hay tal cantidad de modelos aún para un diseño en concreto.

SimpleScalar contiene un conjunto de herramientas para poder simular diseños hardware. Soporta varios repertorios de instrucciones incluyendo Alpha, PowerPC, x86, y ARM. En este proyecto se utilizará la versión que soporta el repertorio de ARM. Para ello es necesario utilizar un compilador cruzado que se ejecute, en este caso en una máquina Intel, y produzca como salida un binario en ARM.

Usa una técnica llamada simulación basada en ejecución que constituye la simulación más cercana a la aplicación. Este proceso requiere reproducir la ejecución de las instrucciones en la máquina a simular. Una alternativa es usar simulación basada en trazas, que emplea determinada información previamente almacenada (trazas) para simular el modelo.

La simulación basada en ejecución tiene una ventaja importante con respecto a la basada en trazas: proporciona acceso a todos los datos producidos durante la ejecución del programa, no como en las trazas que sólo se tiene acceso a ciertos datos, los almacenados en las trazas. Éstos son cruciales para el estudio de posibles optimizaciones. Otra ventaja es que ocupa menos espacio ya que la información almacenada en las trazas ocupa bastante.

Una desventaja de la simulación basada en ejecución es que tiene que leer y decodificar un binario, con la complejidad que ello supone, y luego procesar todas las instrucciones. Esto hace que también sea más lenta que la simulación basada en trazas.

La siguiente figura muestra los simuladores que proporciona el SimpleScalar. Los simuladores que implementa comprenden desde el sim-safe que es el más sencillo hasta el sim-outorder que incluye ejecución especulativa, varios niveles de memoria y ejecución fuera de orden. Todos ellos simulan cada ciclo de reloj que dura el programa.

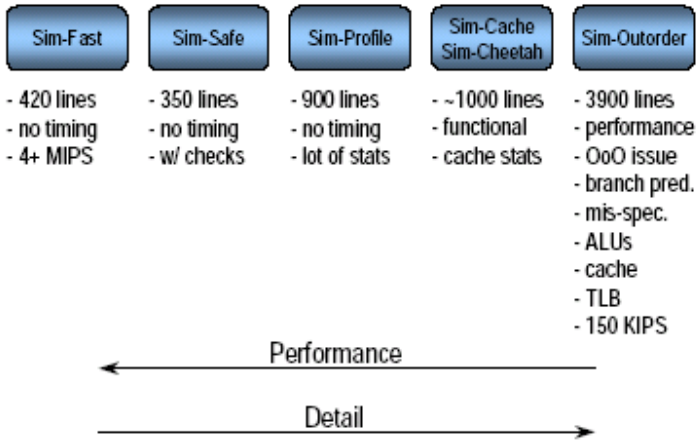

Figura 20: simuladores implementados por el SimpleScalar.

El simulador utilizado en este proyecto ha sido el sim-outorder. Tiene un pipeline de 6 etapas: Búsqueda, Decodificación, Lanzamiento , Ejecucion, Escritura y Commit. En la Figura podemos observar estas etapas junto con las estructuras de datos que son necesarias en cada una.

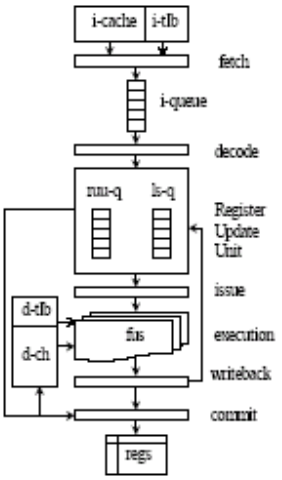

Figura 21 : etapas del pipe del sim-outorder.

En la etapa de búsqueda lee las instrucciones de la caché de instrucciones y las almacena en una cola (IFQ, Instruction Fetch Queue). En la imagen, la i-queue. En esta etapa también predice los saltos.

Las direcciones virtuales de instrucciones y datos se traducen a direcciones reales a través del TLB de instrucciones y de datos respectivamente. El número de instrucciones que se buscan depende de la anchura que se ponga a la etapa, de la capacidad de la IFQ y de la precisión de la predicción de saltos. Es importante resaltar que todas las estructuras de datos, anchuras de etapas y latencias son parametrizables en SimpleScalar.

Las instrucciones disponibles en la IFQ se decodifican y se renombran en la etapa de decodificación y se almacenan en la RUU (Register Update Unit), en la imagen ruu-q. Las instrucciones load y store se dividen en dos: una suma que calcula la dirección efectiva de memoria que es almacenada en la RUU y el Load o Store en sí que se almacena en la LSQ (Load/Store queue), en la imagen ls-q.

La etapa de lanzamiento verifica qué instrucciones de la RUU y de la LSQ están listas para ejecutarse (es decir que tengan sus operandos listos y no tengan dependencias) y las lanza a la unidad funcional correspondiente. El número de instrucciones lanzadas depende del número de instrucciones listas, el número de instrucciones máximo que se pueden lanzar y la disponibilidad de las unidades funcionales.

La etapa de ejecución ejecuta las instrucciones y mantiene cada unidad funcional ocupada durante lo que dura la operación. De las instrucciones de memoria sólo se ejecutan los loads en esta etapa. Los store son ejecutados en la etapa commit cuando la ejecución deja de ser especulativa.

La etapa commit examina la RUU y retira las instrucciones que hayan terminado en orden. Cuando una suma se retira de la RUU, la última entrada de la LSQ también se retira ya que es la otra parte de la instrucción. Cuando se encuentra un salto se prueba a ver si se ha hecho bien la predicción. Si está mal se retiran todas las entradas de la RUU y de la LSQ. Los resultados que no han sido especulados son almacendos en los registros y en memoria.

4.1.1 Modelo de consumo de energía: Wattch

La disipación de energía y el aumento de la temperatura han adquirido gran importancia en los diseños de los procesadores modernos sobre todo en los sistemas empotrados en los que la principal fuente de energía es una batería. Como consecuencia muchos investigadores han desarrollado herramientas para estimar el consumo de energía de las arquitecturas.

Antes de finales de los 90 la mayoría de los modelos de energía operaban a muy bajo nivel y alcanzaban gran precisión calculando una estimación de la energía para los diseños para los que sus desarrolladores habían construido el layout.

Con estas cosas en mente en 1998 se empezó a desarrollar Wattch,. Realiza el análisis del consumo de energía basándose en el uso y la actividad de los distintos componentes como la caché, el banco de registros, etc. La lógica combinacional no se modela, se utilizan constantes.

Al principio se calculan una serie de parámetros físicos teniendo en cuenta diversas características de la estructura a estudiar como puede ser por ejemplo el tamaño o, en el caso de la caché, la asociatividad, etc. Con todos ellos, se calcula el consumo por acceso a dicha estructura utilizando unas ecuaciones previamente desarrolladas.

Para cada uno de los componentes se mira, en cada ciclo, si ha habido un acceso o no y si lo ha habido se incrementan una serie de contadores.

La disipación de energía va cambiando según van variando distintos parámetros como el tamaño de la ventana de instrucciones, el tamaño de caché, el número de instrucciones lanzadas y demás. Finalmente, se pueden llevar a cabo un estudio sobre la disipación de energía para ver dónde se está gastando más y poder mejorarlo.

4.2 Construyendo la scratch

Este proyecto añade al SimpleScalar un nuevo módulo para simular una memoria scratch. Cabe destacar que no se implementa una estructura de datos para almacenar datos concretos, sino que principalmente se centra en contabilizar los accesos que se realizan y calcular su latencia y el consumo de energía asociado a ellos. Esta idea proviene de un estudio detallado del código correspondiente a la memoria caché.

El número de accesos se incrementa si, en una lectura o escritura a la memoria, la dirección accedida está en el rango de direcciones asociado a la scratch. Las direcciones que ocupa la scratch se hallan sumando la dirección de inicio (ya veremos más adelante cómo se calcula) y el tamaño. Las lecturas y escrituras tienen lugar en el sim-outorder en las etapas de commit y lanzamiento. Un punto importante a tener en cuenta es que, si se ha realizado un acceso a la scratch, no se debe comprobar si se ha realizado un acceso a la caché porque es imposible.

La latencia de la scratch es parametrizable. Otros posibles parámetros que acepta son la dirección inicial, el tamaño, el número de puertos y el tamaño de bloque. Una posible ampliación sería tener en cuenta el número de puertos ya que en este proyecto no se ha tenido en cuenta. Se ha considerado que sólo hay uno. Una scratch viene representada por la tupla: dirección inicial, tamaño,

Tambien es importante resaltar que se ofrece la posibilidad de configurar varias memorias scratch para una ejecución.

Para conseguir asignar variables a la scratch se pensaron varias alternativas. Podría haberse modificado el compilador de C para que, además de asignar regiones para variables, por ejemplo, asignase una región de memoria para la scratch y analizase el código para estudiar qué variables fuesen susceptibles de alojarse en ella. Pero para ello habría que cambiar el compilador de C, tarea demasiado complicada.

Otra alternativa que también requería modificar el compilador sería añadir una palabra reservada, por ejemplo **scratch**, que indicase que la variable a la que acompañaba estaba en la scratch. También fue desechada por el mismo motivo.

Al final, se optó por añadir en el código fuente un array de caracteres que simula la memoria scratch. Por cómo está implementado este proyecto, es importante que esta variable sea global, ya diremos más adelante por qué.

Cuando una variable está en la scratch se asigna a una posición de dicho array. Se tomó la determinación de que si una variable era asignada a la scratch, se declaraba como puntero para que quedase más claro que era una referencia a una posición de memoria.

Ahora se explicará cómo se transforma el código para incluir una variable en la memoria scratch. Corresponde al código utilizado por el benchmark de multiplicación de matrices utilizado en este proyecto y descrito en la siguiente sección de esta memoria. Se dan los detalles más relevantes, es decir no se detallan funciones auxiliares como puede ser init, por ejemplo. Originalmente tenemos el código mostrado por el Algoritmo 1:

```
int X[N*N];
int Y[N*N];
int Z[N*N];

int main(){
  int sum1;
  int i,j,k;


  init(X,Y);


  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {

      sum1 = 0;
      for (k=0; k<N; k++) {
        sum1 = sum1 + X[i*N+k] * Y[j+k*N];
      }
      Z[i*N+j] = sum1;
    }
  }
}
```

Algoritmo 1: código original que se quiere transformar para alejar X, Y, Z en la scratch

Se puede ver cómo se declaran tres variables globales X, Y y Z, que se inicializan y posteriormente se opera con ellas. En el caso de que nos interese alojarlas en la scratch el código sería como el mostrado por el algoritmo 2:

```
char SCRATCH1[SC_TAM1];


int main(){
  int sum1;
  int i,j,k;
  int id1,id2,id3;
  int *X;
  int *Y;
  int *Z;

  X = (int *)SCRATCH1;
  Y = (int *)&SCRATCH1[N*N];
  Z = (int *)&SCRATCH1[2*N*N];

  init(X,Y);


  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {

      sum1 = 0;
      for (k=0; k<N; k++) {
        sum1 = sum1 + X[i*N+k] * Y[j+k*N];
      }
      Z[i*N+j] = sum1;
    }
  }


}
```

Algoritmo 2: emplazamiento de X, Y y Z en la scratch.

En este caso, si queremos alojar X, Y y Z en la scratch debemos declararlas locales y como punteros. Luego realizaremos una distribución o mapeo de la memoria según el tamaño de las variables a colocar. En este caso, como queremos colocar toda la matriz entera tenemos la siguiente distribución. La primera posición de la Y vendrá dada por el espacio ocupado por la X, es decir N*N y lo mismo de la Z. Es importante decir que este mapeo es muy importante para que no se solapen las regiones de memoria de una matriz con las de otra matriz.

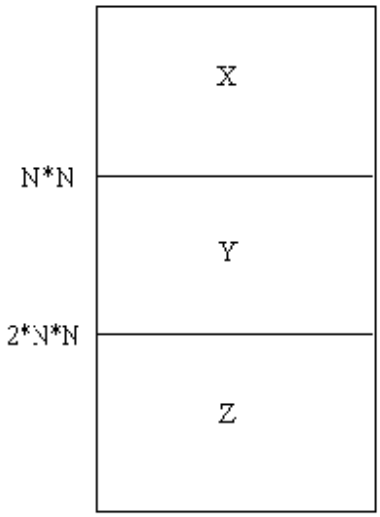

Figura 22: mapeo de la scratch para el código utilizado como ejemplo.

Los pasos necesarios para una simular la memoria scratch requieren primero compilar el código fuente escrito en un fichero .c, luego ejecutar la orden nm que desensambla el binario y devuelve la dirección asignada a cada variable en el código. Por eso, es importante que el array que representa la scratch sea global. Si fuese local estaría en la pila y no se podría hallar su dirección con esta orden. El resultado que nos interesa de esta orden es la lista con todas las variables globales que aparecen en el código fuente y de todas ellas se buscará la dirección asociada a la scratch.

Después, se coge la dirección de esa variable y se incluye en un fichero de configuración en la línea de configuración de la scratch y por último se ejecuta el sim-outorder con las opciones correspondientes.

Se ha utilizado un fichero de configuración para ir más rápido y no tener que escribir siempre los parámetros de la scratch y caché a mano. Debido a la complejidad de este proceso, sobre todo en términos de tiempo, se implementó un script.

El consumo de energía asociado a la scratch también está modelado y está basado en el de la caché pero sin tener en cuenta el gasto asociado a componentes que no existen en la scratch, por ejemplo el array de etiquetas.

Para ello se modificó el fichero power.c que implementa el modelo de energía del SimpleScalar y se añadieron contadores. Se hizo un estudio detallado del código y se modificó para incluir el modelo de la scratch de forma similar al de la caché de datos de primer nivel. Es preciso comentar que en un principio la versión de SimpleScalar de este proyecto no tenía integrado el Wattch y que se tuvo que integrar sobre la marcha.

4.3 Construyendo el DMA.

El principal problema que se presentaba era cómo indicar desde la aplicación que en ese momento se estaba simulando, que se necesitaba hacer una transferencia de memoria principal a scratch.

Para conseguirlo se barajaron varias posibilidades. Una de ellas era implementar las transferencias como una lectura o escritura a una posición de control previamente designada. Para ello hubiera sido necesario el empleo de código ensamblador en la aplicación a simular. Hubiera sido lo más realista y la más rápido hablando en términos de tiempo de ejecución, ya que al fin y al cabo es lo más parecido a lo que hace el DMA. Otra posibilidad era añadir nuevas instrucciones al repertorio que simulasen una transferencia, pero eso suponía demasiados cambios, sobre todo en el compilador, tarea demasiado complicada dado el poco tiempo con el que se contaba.

Al final se optó por la manera más fácil y más rápida, que era utilizar un mecanismo de llamada al sistema, que no es más que una función con un código y unos parámetros. Es decir, se decidió emplear una syscall aprovechando el hecho de que el SimpleScalar proporcionaba un módulo de llamadas al sistema, syscall.c, encargado de establecer la comunicación entre la aplicación y el simulador.

Se añadieron tres nuevas llamadas (una para transferencia lineal, otra para transferencia por bloques y otra para espera) que no existían y se estudió la forma en que se manejaba el paso parámetros para hacerlo adecuadamente.

Los parámetros de las transferencias lineales y por bloques ya se ha comentado previamente. Queda por comentar los parámetros de la espera. Acepta un solo parámetro que es el identificador de la transferencia por la que está esperando. En este proyecto se utiliza un entero que representa el número de transferencia que se va incrementando cada vez que se inicia una transferencia.

Para implementar una transferencia se estima el número de ciclos que va a tardar calculando la latencia de los accesos a memoria que se van a realizar y se los suma al ciclo de reloj actual. La latencia la marca el tiempo que se tarda en leer de memoria principal los datos que queremos transferir. Luego se hace la copia de datos en sí.

La función de espera consulta si el ciclo de reloj en el que acaba la correspondiente transferencia es mayor que el ciclo de reloj actual. Si es mayor crea una variable que representa el número de ciclos a esperar. Mientras esta variable sea positiva se evita que la búsqueda de nuevas instrucciones.

En cuanto al modelo de energía del DMA no se modela ya que se considera despreciable porque únicamente sería un conjunto de registros con una máquina de estados.

5. BENCHMARKS

El benchmarking es una técnica utilizada para medir el rendimiento de un sistema o parte de un sistema, frecuentemente en comparación con algún parámetro de referencia.

Más formalmente puede entenderse que un benchmark es el resultado de la ejecución de un programa informático o un conjunto de programas en una máquina, con el objetivo de estimar el rendimiento de un elemento concreto o la totalidad de la misma, y poder comparar los resultados con máquinas similares. También puede realizarse un "benchmark de software", es decir comparar el rendimiento de un software contra otro.

La tarea de ejecutar un benchmark originalmente se reducía a estimar el tiempo de proceso que lleva la ejecución de un programa (medida por lo general en miles o millones de operaciones por segundo). Con el correr del tiempo, la mejora en los compiladores y la gran variedad de arquitecturas y situaciones existentes convirtieron a esta técnica en toda una especialidad. La elección de las condiciones bajo la cual dos sistemas distintos pueden compararse entre sí es especialmente ardua, y la publicación de los resultados suele ser objeto de candentes debates cuando estos se abren a la comunidad.

En el presente trabajo interesan áreas de aplicación con importancia relevante en los sistemas empotrados. Quizás el procesamiento multimedia pueda ser la más inmediata. Por tanto se introducirá un análisis de los algoritmos de compresión-descompresión para JPEG y MPEG:

- El algoritmo **MPEG** es un grupo formado por varios estándares para el desarrollo de una técnica de codificación y compresión de vídeo/audio digital. Los estándares MPEG permiten la multiplexación de diferentes canales de vídeo, audio y datos, junto con la información temporal necesaria para lograr una reproducción sincronizada, en una sola trama, llamada trama de sistema.

.

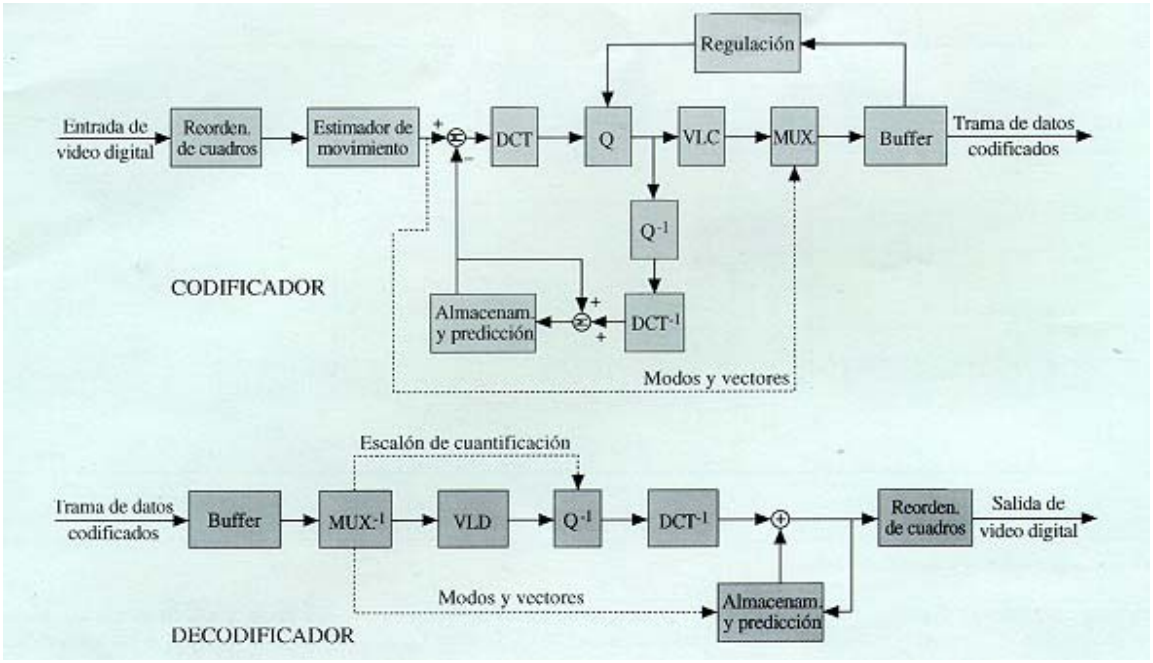

Figura 23: Compresión y descompresión para MPEG

- El algoritmo **JPEG** de compresión de imágenes se compone de varias etapas. Comienza convirtiendo la imagen desde su modelo de color RGB a otro llamado YUV. Después se realiza un submuestreo. Posteriormente, cada componente de la imagen se divide en pequeños bloques de 8x8 píxeles. Cada pequeño bloque se transforma al dominio de la frecuencia mediante un algoritmo de transformación directa del coseno (DCT). Después se produce una cuantización y luego se sigue el mismo proceso de forma inversa mediante una decodificación.

Como se puede observar en ambos procesos se utiliza una transformación directa del coseno y una cuantización (íntimamente relacionado por la multiplicación de matrices). No es de extrañar por tanto que dos de los benchmarks elegidos para testear los algoritmos hayan sido precisamente el DCT y la multiplicación de matrices. El tercer algoritmo escogido ha sido el FIR, que no es más que un tipo de filtros digitales en el que, como su nombre indica, si la entrada es una señal impulso, la salida tendrá un número finito de términos no nulos. Los filtros FIR tienen la gran ventaja de que pueden diseñarse para ser de fase lineal, lo cual hace que presenten ciertas propiedades en la simetría de los coeficientes. Tiene muchísima importancia en aplicaciones de audio. Como una aplicación adicional de la multiplicación de matrices se puede destacar el campo de la cinemática directa dentro del área de la robótica, mediante el uso de matrices de transformación, además y por descontado de la fundamental importancia de esta operación en el álgebra lineal.

A lo largo de las pruebas se toman varias propuestas para ver cuál es la mejor solución a un problema. Para que estuviesen equilibradas, se las dio a cada una de ellas una memoria de 4KB a dividir entre memoria caché de primer nivel y memoria scratch, aunque en algunas se permitió una pequeña desviación. En las pruebas no había memoria caché de segundo nivel.

Organización de los benchmarks.

Normalmente se hicieron los experimentos de acuerdo a uno de los siguientes patrones:

- Utilización de una memoria scratch de 4KB con una pequeña caché de 256B. Esta memoria caché tenía dos conjuntos, con tamaño de bloque dieciséis y grado de asociatividad ocho. En un sistema real no se implementaría, su utilidad es intentar paliar las deficiencias que se presentan con el modelo de scratch que se ha elegido para almacenar escalares, índices de bucles y otros elementos problemáticos, esperando que se comporten posicionándose en ella, es decir, es una solución software que se intenta dar a un problema planteado por las limitaciones de la implementación.

- Utilización de una memoria scratch de 2KB con una caché mediana de 2KB. Esta memoria caché tenía cuatro conjuntos, con tamaño de bloque dieciséis y grado de asociatividad treinta y dos.

- Utilización de una memoria caché de 4KB, sin memoria scratch. Esta memoria caché tenía cuatro conjuntos, con tamaño de bloque treinta y dos y grado de asociatividad treinta y dos. Este patrón se corresponde con el algoritmo original y se utiliza como base para la comparación del rendimiento. Ninguna solución basada en scratch que obtenga un rendimiento inferior a ésta vale para nada.

- Utilización de una memoria caché de 4KB con una pequeña scratch de 256 o 512 bytes. Esta memoria caché tenía cuatro conjuntos, con tamaño de bloque treinta y dos y grado de asociatividad treinta y dos.

- Utilización de prebúsqueda. En realidad esto no es un patrón sino que es una decisión que convive con alguna de las configuraciones con scratch, y realmente es una forma de que el programador otorgue a la scratch una ventaja frente a la caché, del mismo modo que los automatismos de la caché hacen lo propio frente a la scratch. Las soluciones con prebúsqueda son las más complejas porque implican una modificación del algoritmo. En nuestro estudio las técnicas de prebúsqueda han sido utilizadas con arrays, bien por filas o columnas, bien por bloques. Esta última solución por bloques es con diferencia la más compleja pero también es con la que se obtienen mejores resultados cuando se emplea correctamente.

En las soluciones con scratch, se hace necesario conocer el patrón de acceso a los datos del problema. Una matriz a la que se accede por filas tendrá un rendimiento bastante bueno en la memoria caché, no así una a la que se haga por columnas, ya que provocará numerosos fallos de caché con los consiguientes accesos a memoria principal, qué es en realidad lo que se trata de minimizar. Un mismo dato al que se accede de forma repetida también funcionará muy bien en la caché.

Los datos que funcionan mal en la caché son los principales candidatos a permanecer dentro de la scratch. También se debe tener en cuenta el orden de los accesos, interesará más optimizar el acceso a un dato con un orden cúbico que a otro con un orden cuadrático.

Para cada benchmark se mostrará en primer lugar el algoritmo inicial. Después, cuando haya alguna pequeña modificación debida a una prebúsqueda o la utilización de la scratch se intentará exponer teóricamente.

Como ya se comentó previamente, la scratch debe ser declarada como un array de caracteres (bytes). El proceso de ejecución del código fuente utiliza un script que consulta la dirección asignada por el compilador de c a la variable scratch y la escribe sobre el fichero de configuración que corresponde a la ejecución en el simple-sim.

Este script también halla el tamaño asignado a la scratch y lo escribe sobre el fichero de configuración. Para acceder a una posición de la scratch se utiliza un puntero a la misma. Por tanto, a veces hay una pequeña diferencia entre las soluciones basadas en caché o en scratch en la forma en la que se declara un dato: array en caso de las soluciones sin scratch y puntero en caso contrario. Pero en este análisis de los benchmarks nos abstraeremos de estos detalles.

Cuando se utiliza una prebúsqueda se debe reservar un espacio libre o con datos ya no útiles en la memoria scratch. Mientras se van utilizando los datos correspondientes a la zona de uso actual de la scratch se van prebuscando los siguientes datos a usar en esa zona con espacio libre. Esta prebúsqueda implicará el uso del DMA en la mayoría de las ocasiones, utilizando la llamada al sistema 217 para una transferencia lineal y una llamada 219 para una transferencia rectangular. Cuando se quiera hacer uso de los datos prebuscados (que en ese momento pasan a ser, desde el punto de vista de la lógica del programa, los datos de uso actual), se debe hacer anteriormente una llamada al sistema 218 para que espere en caso de que la transferencia correspondiente a esa prebúsqueda aún no haya sido completada.

Limitaciones del entorno de ejecución
El compilador que se utiliza no hace la mejor asignación posible de memoria a las variables. Así, valores muy accedidos como pueden ser los escalares no se posicionan en registros. Por eso en todas las versiones el acceso a los datos representa una pequeña parte de los accesos a memoria totales y así se hace más difícil el análisis de los resultados.

Además, esto nos obliga a tener que asociar una pequeña caché de 256 bytes a la versión con 4KB de scratch, para que esos accesos no sean a memoria principal y se invaliden completamente los resultados del tiempo de ejecución.

La energía consumida también se ve afectada por estas limitaciones. El elevado número de accesos a memoria caché residuales provoca que la mayor parte de la energía sea consumida por esos accesos y por t

Primeros experimentos.

Antes de comenzar las pruebas se probaron distintas posibilidades:

- declaración de variables local o declaración local. Se llegó a la conclusión de que las variables locales daban un tiempo de ejecución menor.

- alojar las variables auxiliares en la scratch o no. Se observó que el tiempo de ejecución aumentaba ligeramente y se decidió no alojar variables tales como enteros en la scratch.

- acceder directamente a la variable scratch o acceder mediante punteros que la referencien. Se comprobó que era bastante mejor el rendimiento cuando se accedía a la scratch a través de punteros.

En cuanto a la distribución de esta memoria, se presenta primero la descripción de las distintas versiones de cada experimento, acompañadas de pseudocódigo en el caso de que sea necesario explicar algún detalle, luego las tablas con los resultados más significativos de cada ejecución, por último, se hace un análisis de los resultados obtenidos y se muestran gráficas relativas al tiempo de ejecución y al consumo total de la memoria.

5.1 Multiplicación de matrices.

El primer benchmark utilizado en el proyecto parte de la base del algoritmo clásico de multiplicación de matrices, que toma dos matrices X e Y de entrada y genera la matriz de salida resultado Z.

Algoritmo original:

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
      sum1 = 0;
      for (k=0; k<N; k++) {
         sum1 = sum1 + X[i,k] * Y[k,j];
      }
      Z[i,j] = sum1;
  }
}
```

Algoritmo 3: código original de multiplicación de matrices.

Como se puede ver, el acceso a los arrays es de orden cúbico para la X y la Y, así como a la variable sum1, y cuadrado para la Z. El patrón de acceso para la X y la Z es por filas, mientras que el de la Y es por columnas. Al dato sum1 se accede en todas las iteraciones del bucle más interno, es decir es un dato al que se accede continuamente.

Para este benchmark se hicieron dos tipos de pruebas: unas con tamaño de los datos (N) diez y otras con cincuenta.

Prueba 1: Con tamaño de los datos (N) diez.

Esta es una prueba en la que todos los datos pueden ser posicionados enn en la scratch, sea de tamaño 2KB o 4KB. Por tanto la opción de prebúsqueda queda descartada.

Versión 1.

Para esta versión se ha utilizado memoria caché de 4KB sin memoria scratch.

El código no se modifica al ser la versión básica que no incorpora scratch, el algoritmo es el original.

Versión 2.

Para esta versión se ha utilizado una memoria scratch de 4KB con una pequeña caché de 256B.

El código se modifica para almacenar variables en la memoria scratch de acuerdo al siguiente razonamiento: al ser N diez y ser X, Y, Z matrices cuadradas cada una de ellas tiene cien elementos. Cada entero ocupa cuatro bytes y por tanto cada una de las matrices ocupa cuatrocientos bytes. Las tres entran completas en la scratch e incluso sobra espacio. De hecho, entrarían completas en una

memoria scratch más pequeña, así que para esta prueba se está desperdiciando memoria, lo que nos da idea de que una solución con una scratch menor y más caché daría mejores resultados, debido a las carencias de nuestra implementación, que penaliza para la scratch el acceso a índices o escalares; de no estar limitados no tendría por qué ser así. La variable sum1 podría meterse en la scratch pero como es un dato al que se accede continuamente se supone que no va a ser reemplazado de la caché.

Versión 3.

Para esta versión se ha utilizado una memoria scratch de 2KB con una caché mediana de 2KB.

El código se modifica para almacenar variables en la scratch de acuerdo al mismo estudio que en el caso anterior. Al ser N diez y ser X, Y, Z matrices cuadráticas cada una de ellas tiene cien elementos. Cada entero ocupa cuatro bytes y por tanto cada una de las matrices ocupa cuatrocientos bytes. Las tres entran completas en la scratch e incluso sobra espacio, aunque menos que en la versión anterior.

Tablas de resultados.

Caché 4KB	Valores
Tiempo de ejecución (ciclos):	115.311
Consumo de memoria scratch(mW*ciclos):	0
Consumo de memoria caché (mW*ciclos) :	58.149
Consumo total de memoria (mW*ciclos) :	58.224
Número de accesos a caché:	19.189
Número de accesos a scratch:	0
Número de accesos a memoria principal:	149
Aciertos de caché:	19.040

scratch 4KB	Valores
Tiempo de ejecución (ciclos):	149.995
Consumo de memoria scratch (mW*ciclos):	1.252
Consumo de memoria caché(mW*ciclos):	11.141
Consumo total de memoria(mW*ciclos):	12.732
Número de accesos a caché:	16.796
Número de accesos a scratch:	2.400
Número de accesos a memoria principal:	678
Aciertos de caché:	116.118

Scratch 2KB y caché 2KB	Valores
Tiempo de ejecución (ciclos):	121.175
Consumo de memoria scratch (mW*ciclos):	752
Consumo de memoria caché (mW*ciclos):	30.902
Consumo total de memoria (mW*ciclos):	31.767
Número de accesos a caché:	16.804
Número de accesos a scratch:	2.400
Número de accesos a memoria principal:	235
Aciertos de caché:	16569

Análisis de resultados.

En primer lugar se observa que, como se anticipaba en el análisis del código, la versión con 4KB de scratch no tiene demasiado sentido: los datos entran por completo en una scratch de 2KB de tamaño y por tanto se está desperdiciando memoria.

El hecho de aumentar el tamaño de la scratch nos ha hecho disminuir innecesariamente el de la caché. En realidad, al estar todos los datos en la scratch, el hecho de que la caché sea más pequeña no debería provocar más accesos a memoria principal; pero por nuestras limitaciones, sí que hay valores como los índices de los bucles, que deben ser almacenados en la caché. La pequeña caché asociada no consigue evitar el aumento de accesos a memoria principal y por eso esta versión es la más lenta, (ver [Figura 24]), tiene seiscientos accesos a memoria por unos ciento cincuenta de las demás. Sin embargo es con diferencia la que menos potencia consume.

Comparemos ahora la versión con una caché de 2KB y una scratch de 2KB con la obtenida con la caché de 4KB, sin scratch. Se observa que la versión con caché es un poco más rápida, provocado igual que antes por unos pocos accesos más a memoria principal. En general, con unos datos tan pequeños, el comportamiento de una caché grande es especialmente bueno, en tiempo de ejecución.

En cuanto a la energía, se comprueba lo predicho al hablar de las limitaciones de la implementación: cuanto más pequeña es la caché menos consume el algoritmo, y el resultado obtenido para la versión con scratch de 4KB es incongruente como se puede apreciar en la [Figura 25] .

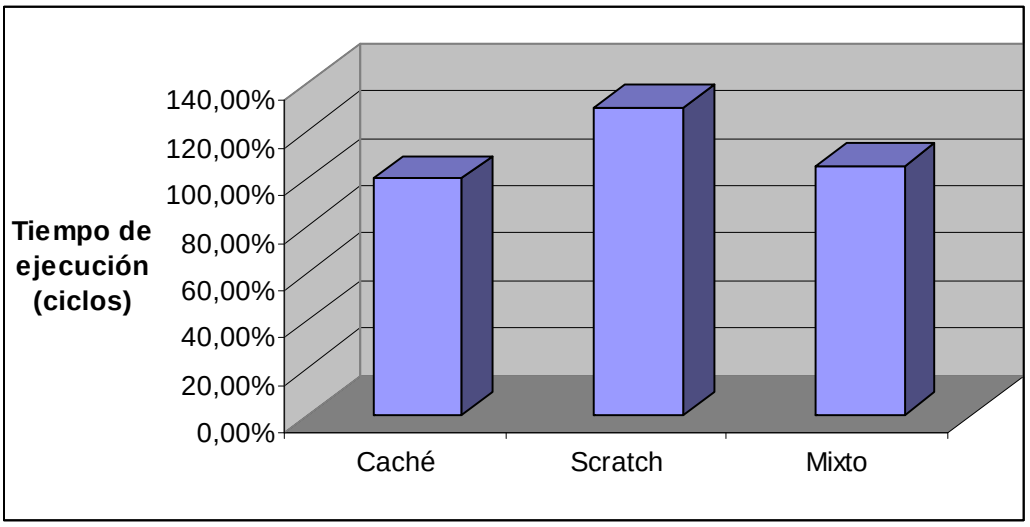

Figura 24: tiempo de ejecución (ciclos) vs configuración de memoria

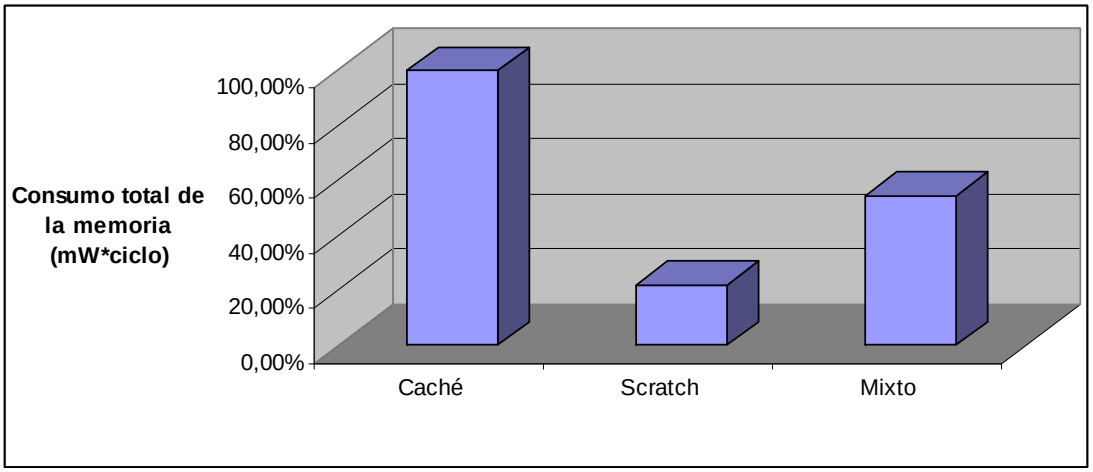

Figura 25: consumo de energía de la memoria (mW*ciclos) vs configuración de memoria

Prueba 2: Con tamaño de los datos (N) cincuenta.

Esta es una versión en la que los datos no entran al completo en la scratch, ni siquiera uno sólo de ellos. Por tanto lo que no tiene mucho sentido es poner un trozo de un dato llenando la scratch y dejar que el resto del programa funcione con caché. Con este tamaño de datos lo que sí tiene mucho sentido es implementar soluciones de prebúsqueda. Así que las soluciones que se darán serán diferentes tipos de prebúsqueda y la basada tan sólo en memoria caché.

Versión 1.

Para esta versión se ha utilizado una memoria caché de 4KB.

El código no se modifica al ser la versión básica que no incorpora scratch, el algoritmo es el original.

Versión 2.

Para esta versión se ha utilizado una memoria scratch de 2KB con una caché mediana de 2KB. Se realiza prebúsqueda para la Y.

El código se modifica para almacenar variables en la memoria scratch de acuerdo al siguiente razonamiento: la variable Y será la que vaya a la scratch. Esto es debido a los patrones de acceso. Como se dijo al presentar el algoritmo, el acceso a la matriz Y es por columnas, mientras que a Z y a X se accede por filas. Por tanto la variable Y se comportará peor en la caché y es la mejor candidata a ir a la scratch.

En cuanto a la prebúsqueda, hay que ir llevándose trozos, en este caso conjuntos de columnas, de Y a la scratch. Primero hay que optimizar el número de columnas a llevarse para minimizar el número de transferencias. La scratch tiene 2048 bytes, de los que 1024 se deben reservar para el dato a prebuscar. Por tanto para el dato efectivo actual sólo se disponen de los otros 1024 restantes. Cada columna tiene cincuenta enteros, y cada uno de ellos ocupa cuatro bytes, por lo que cada columna ocupa doscientos bytes. Así, en el espacio efectivo de la scratch (1024 bytes) se pueden meter cinco columnas de Y, ese es el número de columnas optimizado.

Además, el bucle intermedio que servía para recorrer las columnas se divide en dos. El más externo de ellos recorre los conjuntos de cinco columnas y el más interno hace lo mismo con cada columna dentro de cada conjunto de cinco columnas.

Para explicar la prebúsqueda no se mostrará el código sino un esquema del pseudocódigo:

```
para i= 0,  i< Nfilas,  i++
{
   transferir las cinco primeras columnas de Y a la scratch
   para j= 0,  j<NColumnas,  j=j+5
   {
      espera a que se realice la última transferencia
      prebuscar las siguientes cinco columnas de Y
      para h=0,  h<5,  h++
      {
         sum1=0;
         bucle que realiza el cálculo y deja el resultado en sum1
         Z[índice que toca] = sum1;
      }
   }
}
```

Algoritmo 4: pseudocódigo de prebúsqueda de la Y

Versión 3.

Para esta versión se ha utilizado una memoria scratch de 4KB con una pequeña caché de 256B.

El código se modifica de una forma muy parecida a la versión anterior. Se trata de hacer lo mismo, con la misma variable y utilizando el mismo tipo de prebúsqueda. La única diferencia es que disponemos del doble de espacio en la scratch por lo que el número de columnas que se prebuscan puede ser el doble. Así, en vez de cinco, en cada transferencia se moverán diez columnas. Esta vez tampoco se mostrará el pseudocódigo por ser igual que en la versión anterior.

Versión 4.

Para esta versión se ha utilizado una memoria scratch de 2KB con una caché mediana de 2KB. En esta ocasión se implementará una prebúsqueda para la X y la Y.

La variable Y sigue siendo con diferencia el dato más propicio para ser situado en la scratch. Pero esta vez además se intentará hacer una prebúsqueda para la X.

Para la prebúsqueda, hay que ir llevándose columnas de la Y a la scratch y también filas de la X. Ya que para una fila de la X se recorren todas las columnas, tan sólo se mantendrá una fila de X en la scratch simultáneamente. El mantener esa fila en la scratch hace que el número de columnas que se puedan estar situadas allí sea una menos que cuando no se hacía; antes eran cinco y ahora cuatro.

Al igual que en ejemplo anterior, el bucle intermedio que servía para recorrer las columnas se divide en dos. El más externo de ellos recorre los conjuntos de cuatro columnas y el más interno hace lo mismo con cada columna dentro de cada conjunto de cuatro columnas.

Sin embargo el bucle externo que servía para recorrer las filas no tiene por qué dividirse en dos, ya que al ser el número de filas de cada conjunto de filas tan sólo uno, no tiene sentido recorrer el conjunto de filas y luego cada fila dentro de cada conjunto.

Del mismo modo que antes, para explicar la prebúsqueda no se pondrá el código sino un esquema del pseudocódigo:

```
transferir la primera fila de X a la scratch
para i= 0,  i< N,  i++
{
   transferir las cuatro primeras columnas de Y a la scratch
   esperar a que se realice la última transferencia de la X
   prebuscar la siguiente fila de X
   para j= 0,  j<N,  j=j+4
   {
      espera a que se realice la última transferencia de la Y
      prebuscar las siguientes cuatro columnas de Y
      para h=0,  h<4,  h++
      {
         sum1=0;
         bucle que realiza el cálculo y deja el resultado en sum1 (como siempre)
         Z[índice que toca] = sum1;
      }
   }
}
```

Algoritmo 5: Pseudocódigo de prebúsqueda para X y para Y.

Versión 5.

En esta versión se ha utilizado una memoria caché de 4KB sin scratch. Se implementa un algoritmo de multiplicación por bloques.

Debido a esto, se cambia completamente el código. La idea de este algoritmo es utilizar un bloque de los datos en todos los lugares donde sea necesario para obtener la solución final aunque eso implique dejar resultados a medio calcular. Cuando el algoritmo termina todos los resultados están finalizados.

El algoritmo convierte los tres bucles originales en seis. Para filas y columnas está muy claro: el bucle externo recorre los conjuntos de filas y columnas y el interno cada fila o columna dentro de cada conjunto de filas o columnas. Sin embargo, el bucle relativo a la k desplazamiento también se transforma en dos. Puede resultar algo más complejo de entender. Sin embargo, al entender la k como un desplazamiento, de columna en una fila fija para la primera matriz multiplicando y de fila en una columna fija para la segunda matriz multiplicador, también se podría hablar de un bucle externo para recorrer un conjunto de desplazamientos y de uno interno para cada desplazamiento dentro de ese conjunto. La dimensión del bloque debe intentar optimizarse para cada algoritmo en concreto.

El pseudocódigo del algoritmo de multiplicación de matrices por bloques se muestra a continuación:

```
para i0=0, i0<N, i0= i0 + dim_bloque
{
   para j0=0, j0<N, j0= j0 + dim_bloque
   {
      para k0=0, k0<N, k0= k0 + dim_bloque
      {
         para i=i0, i<(i0+dim_bloque), i++
         {
            para j=k0, j<(j0+dim_bloque), j++
            {
               para k=k0, k<(k0+dim_bloque), k++
               {
                  Z[i,j]= Z[i,j] + X[i,k] * Y[k,j];
               {
            }
         }
      }
   }
}
```

Algoritmo 6: pseudocódigo original de multiplicación por bloques.

Versión 6.

En esta versión se ha utilizado una memoria scratch de 2KB con una caché mediana de 2KB. De nuevo se implementa un algoritmo de multiplicación por bloques. El código toma la base de la versión anterior aunque utiliza prebúsqueda. Así es, en este caso se hará uso de las transferencias por bloques con lo que se espera mejorar el rendimiento.

Al analizar la prebúsqueda, en primer lugar hay que considerar el tamaño máximo de los datos que pueden ser almacenados en la scratch. La scratch tiene 2048 bytes, de los que 1024 se deben reservar para los datos a prebuscar. Por tanto para los datos efectivos actuales sólo se dispone de los otros 1024 restantes. Cada dato ocupa cuatro bytes por lo que se dispone de espacio para 256 elementos. Al ser tres las matrices, sólo se dispone de espacio para 85 elementos en cada una de ellas. Como estamos hablando de bloques cuadrados la dimensión máxima de los mismos será la raíz cuadrada por defecto de 85 es decir nueve.

Este es pseudocódigo del algoritmo incluyendo la prebúsqueda:

```
transferir el primer bloque de X,Y,Z a la scratch
para i0=0, i0<N, i0= i0 + dim_bloque
{
   para j0=0, j0<N, j0= j0 + dim_bloque
   {
      esperar a que se realice la última transferencia de la Z
      prebuscar el siguiente bloque de Z
      para k0=0, k0<N, k0= k0 + dim_bloque
      {
         esperar a que se realice la última transferencia de la X
         prebuscar el siguiente bloque de X
         esperar a que se realice la última transferencia de la Y
         prebuscar el siguiente bloque de la Y
         para i=i0, i<(i0+dim_bloque), i++
         {
            para j=k0, j<(j0+dim_bloque), j++
            {
               para k=k0, k<(k0+dim_bloque), k++
               {
                  Z[i,j]= Z[i,j] + X[i,k] * Y[k,j];
               {
            }
         }
      }
   }
}
```

Algoritmo 7: pseudocódigo de multiplicación por bloques con prebúsqueda

Versión 7.

Para esta versión se ha utilizado una memoria scratch de 4KB con una pequeña caché de 256B, con prebúsqueda por bloques. El código implementado es muy parecido al de la versión anterior. Se trata de hacer lo mismo, la única diferencia es que disponemos del doble de espacio en la scratch. Efectuando los mismos cálculos que antes se obtiene una dimensión máxima de bloque trece.

Tablas de resultados.

caché 4KB	Valores
Tiempo de ejecución (ciclos):	8.669.557
Consumo de memoria scratch (mW*ciclos) :	0
Consumo de memoria cache (mW*ciclos):	5.147.955
Consumo total de memoria (mW*ciclos):	5.157.052
Numero de accesos a cache:	1.698.790
Numero de accesos a scratch:	0
Numero de accesos a memoria principal:	18.194
Aciertos de cache:	1.680.596

caché 2KB y scratch 2KB. Prebúsqueda Y	Valores
Tiempo de ejecución (ciclos):	6.060.880
Consumo de memoria scratch(mW*ciclos) :	78.780
Consumo de memoria cache (mW*ciclos):	2.933.823
Consumo total de memoria (mW*ciclos):	3.014.028
Numero de accesos a cache:	1.595.341
Numero de accesos a scratch:	250.000
Numero de accesos a memoria principal:	2.851
Aciertos de cache:	1.592.490

scratch 4KB. Prebúsqueda Y	Valores
Tiempo de ejecución (ciclos):	9.296.681
Consumo de memoria scratch (mW*ciclos):	130.513
Consumo de memoria cache (mW*ciclos):	1.051.978
Consumo total de memoria (mW*ciclos):	1.202.236
Numero de accesos a cache:	1.585.829
Numero de accesos a scratch:	250.000
Numero de accesos a memoria principal:	39.494
Aciertos de cache:	1.546.335

caché 2KB y scratch 2KB. Prebúsqueda X, Y	Valores
Tiempo de ejecución (ciclos):	5.351.835
Consumo de memoria scratch (mW*ciclos):	95.419
Consumo de memoria cache(mW*ciclos):	2664230
Consumo total de memoria (mW*ciclos):	2.760.890
Numero de accesos a cache:	1.448.743
Numero de accesos a scratch:	302.802
Numero de accesos a memoria principal:	2.481
Aciertos de cache:	1.446.262

caché 4KB. Por bloques	Valores
Tiempo de ejecución (ciclos):	7.398.356
Consumo de memoria scratch (mW*ciclos):	0
Consumo de memoria cache(mW*ciclos) :	5.848.121
Consumo total de memoria (mW*ciclos):	5.848.293
Numero de accesos a cache:	1.929.840
Numero de accesos a scratch:	0
Numero de accesos a memoria principal:	344
Aciertos de cache:	1.929.496

caché 2KB y scratch 2KB. Por bloques. Prebúsqueda.	Valores
Tiempo de ejecución (ciclos):	6.370.896
Consumo de memoria scratch (mW*ciclos):	124.893
Consumo de memoria cache(mW*ciclos):	2.329.445
Consumo total de memoria (mW*ciclos):	2.455.122
Numero de accesos a cache:	1.266.695
Numero de accesos a scratch:	396.333
Numero de accesos a memoria principal:	1.578
Aciertos de cache:	1.446.262

scratch 4KB. Por bloques. Prebúsqueda	Valores
Tiempo de ejecución (ciclos):	5.816.230
Consumo de memoria scratch (mW*ciclos):	226.597
Consumo de memoria cache(mW*ciclos) :	911.574
Consumo total de memoria (mW*ciclos):	1.139.346
Numero de accesos a cache:	1.374.175
Numero de accesos a scratch:	434.051
Numero de accesos a memoria principal:	2351
Aciertos de cache:	1.371.824

Análisis de resultados

En la versión con scratch que incluye la prebúsqueda para la Y, la mejor solución es mezclar scratch y caché. De nuevo, la pequeña caché asociada a la versión con una caché grande es incapaz de paliar el daño que provocan los índices, escalares y demás elementos conflictivos. La versión con caché de 4KB accede mucho a memoria principal debido a el resto de datos (X y Z) no se almacenan en ningún momento en la scratch. Por eso es incluso peor que la versión simple con caché de 4KB, no tiene ningún sentido que sea implementada.

Este mismo análisis es válido para la versión que incluye prebúsqueda para la X y la Y, por eso mismo no se ha implementado versión con sólo scratch, pues cabría esperar peores resultados que para la versión mixta.

Como puede verse en la [Figura 26] varias de estas versiones ofrecen un resultado peor que la versión simple con sólo caché por lo que son claramente inaceptables. Esto demuestra que no todas las prebúsquedas son válidas, una prebúsqueda que no tenga claros sus objetivos puede degradar el rendimiento. En estos dos casos, por ejemplo, no vale echar toda la culpa a nuestra implementación. Al haberse utilizado datos de tamaño lo suficientemente grande como para que no pudiesen ser posicionados por entero en la scratch, si realmente se quiere obtener un buen rendimiento para una versión que no tiene caché, se debe hacer prebúsqueda para todos los datos. Así es, con que uno de los datos, como podría ser la Z, no se sitúe en todo momento en la scratch el número de accesos a memoria principal será muy elevado y por tanto el tiempo de ejecución del algoritmo aumentará.

Sin embargo en la versión que incluye prebúsqueda por bloques la mejor solución es utilizar una scratch grande con una pequeña caché. En este caso se ha utilizado prebúsqueda para todos los datos y eso se deja notar en el rendimiento. En realidad se obtienen unos pocos ciclos de ejecución más que para la versión con caché de 4KB por bloques, pero prácticamente es igual, y si se tiene en cuenta el daño que provocan las limitaciones de nuestra implementación a las versiones con scratch, se podría apostar por unos mejores resultados para esta versión en un entorno de ejecución no tan limitado.

Si se comparan entre sí las diversas técnicas de prebúsqueda, se comprueba que cuantos más datos se intentan prebuscar mejor es el resultado obtenido, así, la prebúsqueda con peor rendimiento es la que sólo se ocupa de la X, después la que se ocupa de la X y la Y, y la mejor es la que prebusca todos los datos por bloques.

Otro dato interesante es que los dos primeros tipos de prebúsquedas, que se han comprobado no válidos para una versión sólo con scratch, sí que funcionan bastante bien en la versión que mezcla caché y scratch, sin duda porque la scratch recoge las ventajas de la prebúsqueda y la caché controla los datos que no se posicionan en la scratch.

En cuanto a la energía, nos encontramos con los resultados de siempre: las versiones con caché grande consumen más como puede apreciarse en la [Figura 27], resultado que es válido al comparar cualquier versión excepto las que tienen scratch de 4KB en las que se observa demasiada ganancia para ser coherente.

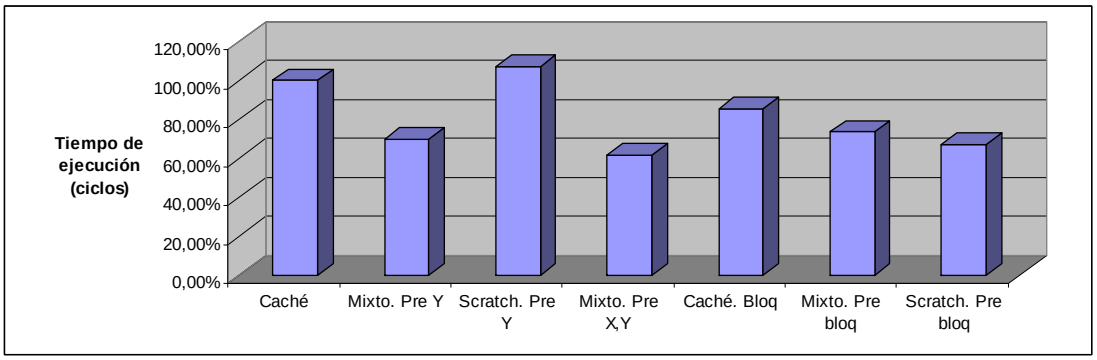

Figura 26: tiempo de ejecución (ciclos) vs configuración de memoria

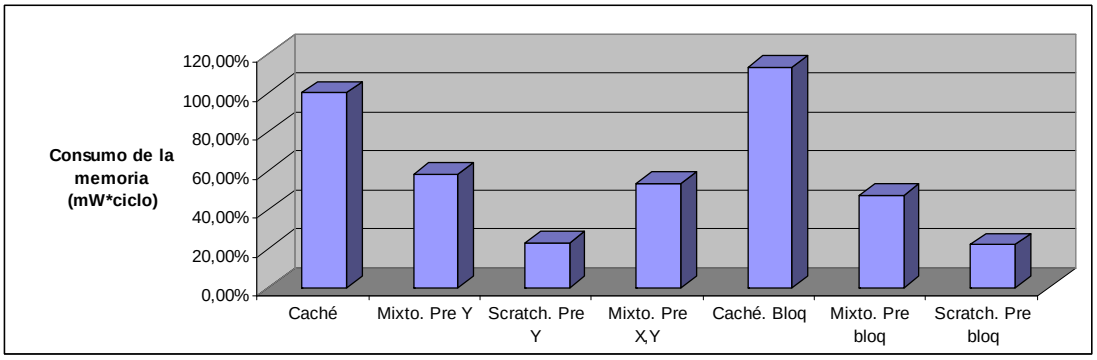

Figura 27: consumo de energía de la memoria (mW*ciclos) vs configuración de memoria

5.2 FIR

El primer benchmark utilizado en el proyecto parte de la base del algoritmo FIR.

FIR es un acrónimo en inglés para Finite Impulse Response o Respuesta finita al impulso. Se trata de un tipo de filtros digitales en el que, como su nombre indica, si la entrada es una señal impulso, la salida tendrá un número finito de términos no nulos.

Los filtros FIR tienen la gran ventaja de que pueden diseñarse para ser de fase lineal, lo cual hace que presenten ciertas propiedades en la simetría de los coeficientes. Este tipo de filtros tiene especial interés en aplicaciones de audio. Además son siempre estables.

Algoritmo original:

```
for (i1=0; i1<=N1-N2; i1++)
{
   y[i1] = 0.0;
   for (i2=0; i2<N2; i2++)
   {
      y[i1] += w[i2]*x[i1+i2];
   }
}
```

Algoritmo 8: código original del FIR

Siendo W de tamaño N2 (actúa de máscara), Y de tamaño N1 y X de tamaño N1+N2.

Como se puede ver, el acceso a los arrays es del orden cuadrático para todos. El patrón de acceso para Y es muy regular, ya que se accede al elemento i1 un número N2 de veces seguidas. W tiene un acceso muy irregular puesto que se accede a un elemento cada vez empezando por el principio hasta el final y se vuelve a comenzar. El patrón de acceso de la X es algo extraño. Primero se accede a la serie: 0..N2-1, luego a 1..N2, luego a la 2..N2+1, y así sucesivamente.

Para este benchmark se harán dos tipos de pruebas: unas con tamaño de los datos (N1) ciento veintiocho y otras con mil veinticuatro. El valor de N2 (tamaño de la máscara) siempre es treinta y dos.

Prueba 1: Con tamaño de los datos (N1) mil veinticuatro.

Esta es una versión en la que los datos no entran al completo en la scratch, tan sólo la máscara W. Por tanto lo que no tiene mucho sentido es poner un trozo de un dato (exceptuando la máscara) llenando la scratch y dejar que el resto del programa funcione con caché. Con este tamaño de datos lo que sí tiene mucho sentido es implementar soluciones de prebúsqueda. Además también se verán soluciones con una pequeña scratch adicional a la caché en la que se puedan guardar algunos datos, como la máscara. Así que las soluciones que se darán serán diferentes tipos de prebúsqueda y la basada en memoria caché, con una pequeña scratch en algunos casos.

Versión 1.

Para esta versión se ha utilizado una memoria caché de 4KB, sin memoria scratch.

El código no se modifica al ser la versión básica que no incorpora scratch, el algoritmo es el original.

Versión 2.

Para esta versión se ha utilizado una memoria caché de 4KB con una pequeña scratch de 256B.

El código se modifica ligeramente de forma que la W se almacena durante todo el programa en la scratch. Nótese que W ocupa exactamente 256B, ya que tiene 32 elementos y cada uno de ellos ocupa 8B.

Versión 3.

Para esta versión se ha utilizado una memoria caché de 4KB con una pequeña scratch de 512B.

Se implementa una prebúsqueda para la X. Esto es debido a los patrones de acceso. Como se dijo al presentar el algoritmo, el acceso al vector Y es muy regular, por lo que W y X son mejores candidatos a ocupar la scratch.

Lamentablemente, en este experimento no podemos utilizar prebúsqueda para el vector X mientras se mantiene la máscara continuamente en la scratch, debido a que la scratch debería tener al menos 513B para hacer eso posible. Eso es porque W ocupa 256B y la prebúsqueda requiere 257B.

Para implementar esta prebúsqueda, primero hay que estudiar el patrón de acceso de X. Como se dijo al presentar el algoritmo, el acceso a X es de la forma: primero se accede a la serie: 0..N2-1, luego a 1..N2, luego a la 2..N2+1, y así sucesivamente, hasta que el algoritmo termine. La idea en este caso es hacer una prebúsqueda algo diferente a como la hacíamos otras veces. Con sólo un byte se podrían prebuscar datos de la forma correcta. Así es, bastaría con hacer una transferencia de un dato en cada iteración. En la segunda iteración, los datos son los mismos que en la primera exceptuando el dato N2 el lugar del dato 0. En la tercera, los datos son los mismos que en la segunda exceptuando el dato N+1. El problema al seguir este esquema es que los datos van quedando desordenados en la scratch. En el algoritmo habrá que implementar alguna técnica para llevar la cuenta del desplazamiento relativo del primer elemento en la scratch.

En cuanto a la transferencia del dato, no se invoca al DMA, es un desperdicio hacerlo por un solo dato. Se utiliza el DMA para la transferencia inicial y nada más.

El desplazamiento del dato en la scratch se implementa utilizando la operación módulo, así como la suma de los índices de los dos bucles. Intuitivamente, los dos índices que representan el acceso a un dato mayor sobre una región de memoria pequeña deben llevar un módulo para hacer el acceso posible. En cuanto a esa operación módulo se ha intentado optimizar sustituyéndola por operaciones alternativas y menos costosas en la medida de lo posible para no desvirtuar el rendimiento del algoritmo.

Como ya se ha hecho otras veces, para explicar el esquema de prebúsqueda no se mostrará el código sino un esquema en pseudocódigo:

```
transferir los treinta y dos primeros datos de X a la scratch
bucle externo
   inicializar el dato a procesar en Y a cero
   desp= (desp+1) modulo (N2+1)
   transferir el siguiente dato de X a la scratch
   bucle interno
   {
      sumadeindices= (indice bucle externo+indice bucle interno) modulo (N2+1)
      dato a procesar en Y= dato a procesar de W * X[sumadeindices]
   }
}
```

Algoritmo 9: pseudocódigo de prebúsqueda para la X.

Versión 4.

Para esta versión se ha utilizado una scratch de 2KB y una caché de 2KB.

Se utiliza la misma prebúsqueda para la X que en la versión anterior, por tanto, el código se modifica de acuerdo con el patrón de estudio anterior. Pero en esta ocasión se dispone de espacio suficiente en la scratch para, al mismo tiempo que se hace la prebúsqueda para la X, almacenar la máscara W.

Para este tipo de prebúsqueda no se implementará una solución con una scratch grande. Esto es debido a que esta prebúsqueda no hace ningún tipo de estudio para la variable Y, por tanto, no se puede almacenar en la scratch (no entra entera) y no tiene sentido reducir el tamaño de la memoria caché si no se necesita más scratch, lo único que se logrará es que los accesos a la variable Y degraden el comportamiento del algoritmo.

Tablas de resultados.

caché 4KB	Valores
Tiempo de ejecución (ciclos):	2.720.391
Consumo de memoria scratch (mW*ciclos):	0
Consumo de memoria cache (mW*ciclos) :	2.222.990
Consumo total de memoria (mW*ciclos) :	8.244.103
Numero de accesos a cache:	733.465
Numero de accesos a scratch:	0
Numero de accesos a memoria principal:	646
Aciertos de cache:	732.819

caché 4KB y scratch 256B	Valores
Tiempo de ejecución (ciclos):	2.228.207
Consumo de memoria scratch (mW*ciclos):	4.818
Consumo de memoria cache (mW*ciclos):	2.126.165
Consumo total de memoria (mW*ciclos):	2.131.300
Numero de accesos a cache:	701.620
Numero de accesos a scratch:	31.904
Numero de accesos a memoria principal:	633
Aciertos de cache:	700.987

caché 4KB y scratch 512B. Prebúsqueda X	Valores
Tiempo de ejecución (ciclos):	2.545.910
Consumo de memoria scratch (mW*ciclos) :	6.017
Consumo de memoria cache (mW*ciclos) :	2.358.370
Consumo total de memoria (mW*ciclos) :	2.364.708
Numero de accesos a cache:	778.246
Numero de accesos a scratch:	32.801
Numero de accesos a memoria principal:	642
Aciertos de cache:	777.604

caché 2KB y scratch 2KB. Prebúsqueda X	Valores
Tiempo de ejecución (ciclos):	2.566.028
Consumo de memoria scratch (mW*ciclos) :	20.349
Consumo de memoria cache (mW*ciclos) :	1.372.710
Consumo total de memoria (mW*ciclos):	1.393.689
Numero de accesos a cache:	746.446
Numero de accesos a scratch:	64.577
Numero de accesos a memoria principal:	1.240
Aciertos de cache:	745.206

Análisis de resultados

La versión con caché a la que se la añade una pequeña scratch para almacenar la máscara funciona mejor que la versión con cache, es más rápida y consume menos.

La versión que con caché y scratch de 2KB, que también incluye prebúsqueda, obtiene un resultado algo superior para el tiempo de ejecución con un consumo mucho menor.

Este resultado podría interpretarse como que en realidad es mejor almacenar todo el tiempo W en la scratch (como se hace en la segunda versión) que intentar una prebúsqueda (en realidad es lo que ocurre, los resultados lo demuestran). Pero que en este caso concreto la prebúsqueda no haya obtenido mejores resultados no quiere decir que no sea una técnica válida sino simplemente que a lo mejor este tipo de prebúsqueda no merecía la pena, es decir, que lo que se perdía era más que lo que se ganaba. Aunque en realidad alguna mejora si que hay, porque es mejor que la versión que sólo utiliza caché, tanto en tiempo de ejecución como en consumo energético. De hecho es en el consumo donde esta versión obtiene mejores resultados que todas las demás al optimizar el acceso a la scratch.

Como se aprecia en la [Figura 29] la versión que obtiene un mejor resultado en tiempo de ejecución, la que añade a la caché una pequeña scratch en la que se posiciona la máscara, es una solución bastante interesante y muy simple, aunque se debe recordar que implementa una solución con algo más de memoria que las demás por lo que no es una comparación absolutamente justa.

En cuanto al consumo de energía, como se puede ver en la [Figura 30],la versión con caché de 2KB y scratch de 2KB es notablemente la mejor Era de esperar puesto que es la versión con un mayor número de accesos a la scratch.

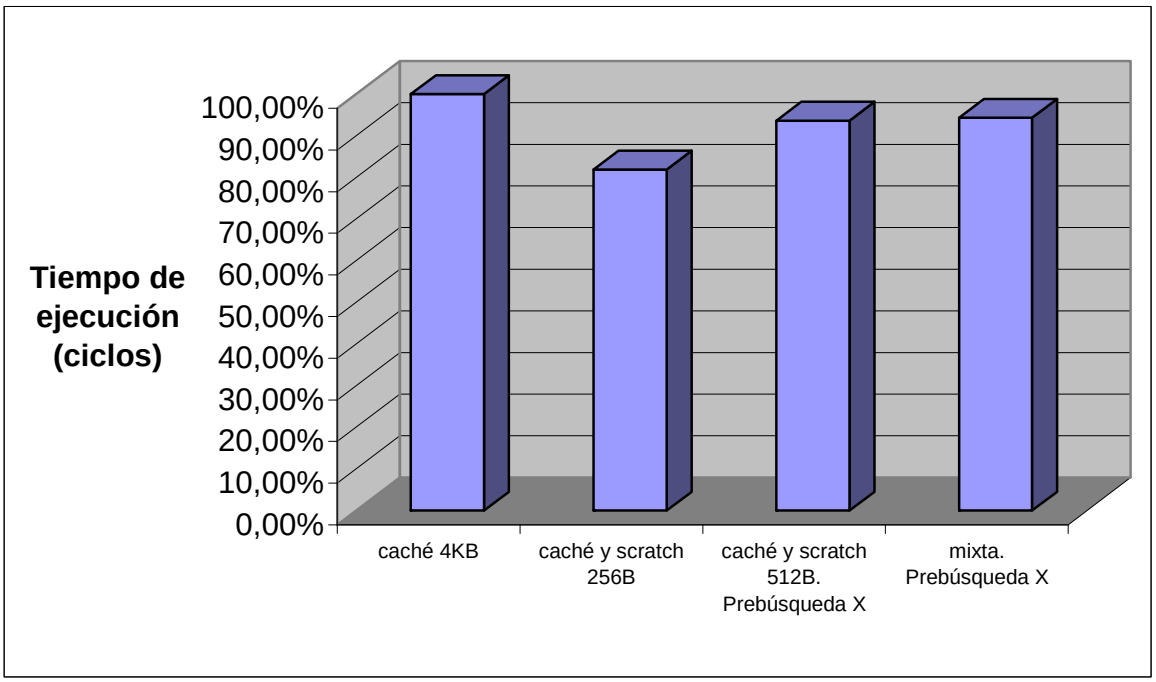

Figura 28: tiempo de ejecución (ciclos) vs configuración de memoria

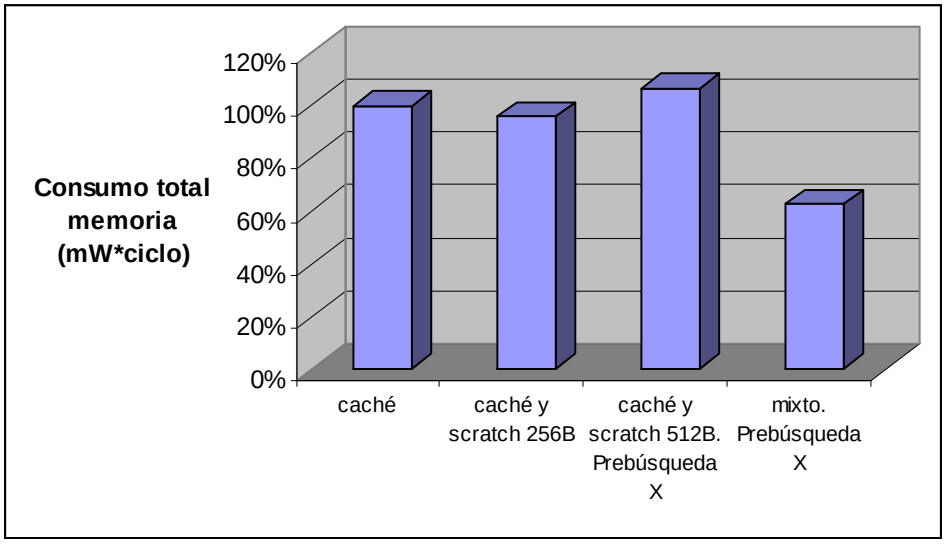

Figura 29: consumo de energía de la memoria (mW*ciclos) vs configuración de memoria

Prueba 2: Con tamaño de los datos (N1) ciento veintiocho.

Esta es una prueba en gran parte de los datos pueden entrar en la scratch, sea de tamaño 2KB o 4KB. Por tanto, la opción de prebúsqueda podría descartarse aunque se implementará en alguna configuración con una pequeña scratch asociada a una caché mayor.

Versión 1.

Para esta versión se ha utilizado una memoria caché de 4KB sin memoria scratch.

El código no se modifica al ser la versión básica que no incorpora scratch, el algoritmo es el original.

Versión 2.

Para esta versión se ha utilizado una memoria scratch de 2KB con una caché mediana de 2KB.

El código se modifica ligeramente de acuerdo con el siguiente estudio: al ser N1 ciento veintiocho y N2 treinta y dos, y los elementos de los vectores de tipo float, X (de tamaño N1+N2) ocupa 1280B y W (de tamaño N2) ocupa 256B. Por tanto no sobran otros 1024B para poner Y.

En cualquier caso se colocan en la scratch los dos vectores con un patrón de acceso más irregular. Como se vio en el análisis del algoritmo, Y tiene un patrón de acceso muy regular y se puede dejar en memoria caché ya que dará un buen rendimiento.

Versión 3.

Para esta versión se ha utilizado una memoria scratch de 4KB con una pequeña caché de 256B.

El código se modifica de acuerdo al mismo estudio que en el caso anterior. Al ser N1 ciento veintiocho y N2 treinta y dos, y los elementos de los vectores de tipo float, X (de tamaño N1+N2) ocupa 1280B y W (de tamaño N2) ocupa 256B. Pero en esta ocasión sí que sobran otros 1024B para poner Y. Por tanto todas las variables serán almacenadas por completo en la scratch.

Versión 4.

Para esta versión se ha utilizado una memoria cache de 4KB con una pequeña scratch de 256B.

El código se modifica según un estudio ya hecho para la primera prueba con tamaño de datos mil veinticuatro. Sería el código original, sólo cambia el hecho de que la W se almacena durante todo el programa en la scratch. Nótese que W ocupa exactamente 256B, ya que tiene 32 elementos y cada uno de ellos ocupa 8B.

Tablas de resultados.

Caché 4KB	Valores
Tiempo de ejecución (ciclos):	349.855
Consumo de memoria scratch (mW*ciclos) :	0
Consumo de memoria cache (mW*ciclos) :	250.984
Consumo total de memoria (mW*ciclos):	251.109
Numero de accesos a cache:	82.823
Numero de accesos a scratch:	0
Numero de accesos a memoria principal:	250
Aciertos de cache:	82.573

Caché 4KB y scratch 256B	Valores
Tiempo de ejecución (ciclos):	291.793
Consumo de memoria scratch (mW*ciclos) :	488
Consumo de memoria cache (mW*ciclos) :	241.344
Consumo total de memoria (mW*ciclos):	241.953
Numero de accesos a cache:	79.642
Numero de accesos a scratch:	3.232
Numero de accesos a memoria principal:	241
Aciertos de cache:	79.401

caché 2KB y scratch 2KB	Valores
Tiempo de ejecución (ciclos):	358.155
Consumo de memoria scratch (mW*ciclos) :	2.036
Consumo de memoria cache (mW*ciclos):	141.194
Consumo total de memoria (mW*ciclos):	143.378
Numero de accesos a cache:	76.778
Numero de accesos a scratch:	6.464
Numero de accesos a memoria principal:	386
Aciertos de cache:	76.392

Scratch 4KB	Valores
Tiempo de ejecución (ciclos):	717.575
Consumo de memoria scratch (mW*ciclos):	6.666
Consumo de memoria cache (mW*ciclos):	46.469
Consumo total de memoria (mW*ciclos):	56.150
Numero de accesos a cache:	70.052
Numero de accesos a scratch:	12.769
Numero de accesos a memoria principal:	6.029
Aciertos de cache:	64.023

Análisis de resultados.

La versión con caché a la que se la añade una pequeña scratch para almacenar la máscara funciona mejor.

La versión con caché y scratch mezcladas da un resultado muy similar a la de la versión que almacena la máscara en la scratch; mientras que la versión con una pequeña caché y una scratch grande es con diferencia la que peor rendimiento ofrece.

En estos datos hay un resultado que puede parecer desconcertante. En efecto, para la prueba con una scratch grande, si todos los datos están almacenados por entero en la scratch, el rendimiento debería ser óptimo, y sin embargo el tiempo de ejecución es del orden del doble del hallado para el resto de las pruebas. A pesar de eso, al analizar los resultados se puede observar la causa, que no es otra que los seis mil accesos a memoria principal, que en el caso del resto de versiones se mantiene entre doscientos y trescientos. Evidentemente, en este caso no se ha logrado que la pequeña caché asociada para intentar evitar los fallos asociados a índices o escalares consiga evitar la degradación del rendimiento.

Exceptuando esto, estos resultados reafirman los análisis obtenidos en la prueba uno con un tamaño de los datos mayor. Sin duda, la mejora más simple y más efectiva es añadir a la caché una pequeña scratch en la que se posiciona un dato relativamente pequeño que va a ser accedido con un patrón de acceso muy irregular, continuamente a lo largo de todo el algoritmo.

Este análisis se reafirma en la versión con caché y scratch mezcladas. Obtiene mejor resultado que la que sólo tiene caché, tanto en consumo energético como en ciclos de ejecución (ver [Figura 30] y [Figura 31]). Esta mejora se debe a los accesos a memoria scratch que evitan accesos a caché, ya que en el resto de datos los resultados son muy similares para las dos versiones, lo que confirma que en el momento en que se consiguen paliar los accesos a memoria principal provocados por las limitaciones de nuestra implementación, la ejecución con scratch ofrece mejores resultados.

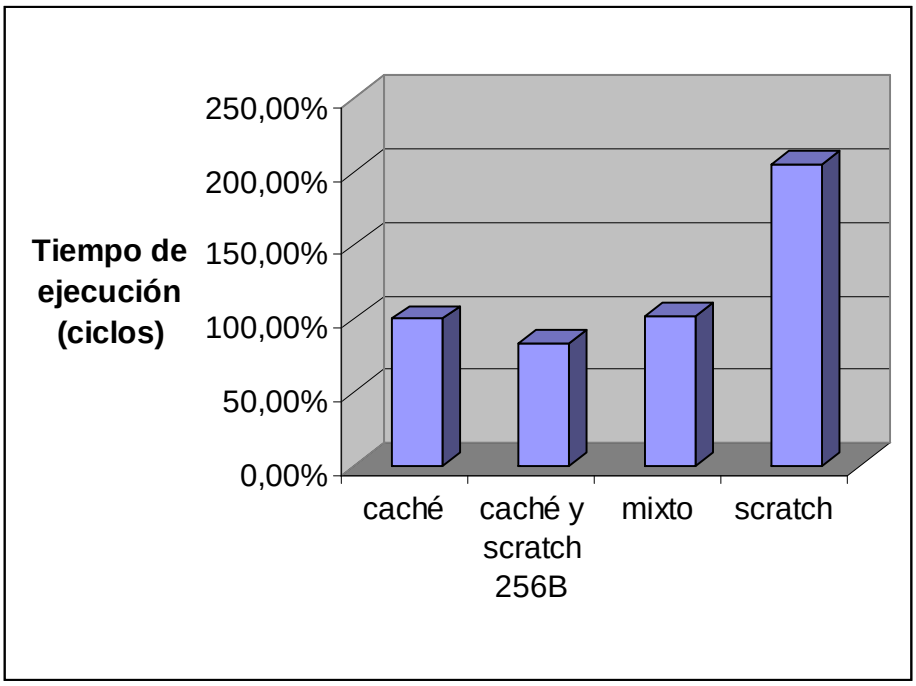

Figura 30: tiempo de ejecución (ciclos) vs configuración de memoria

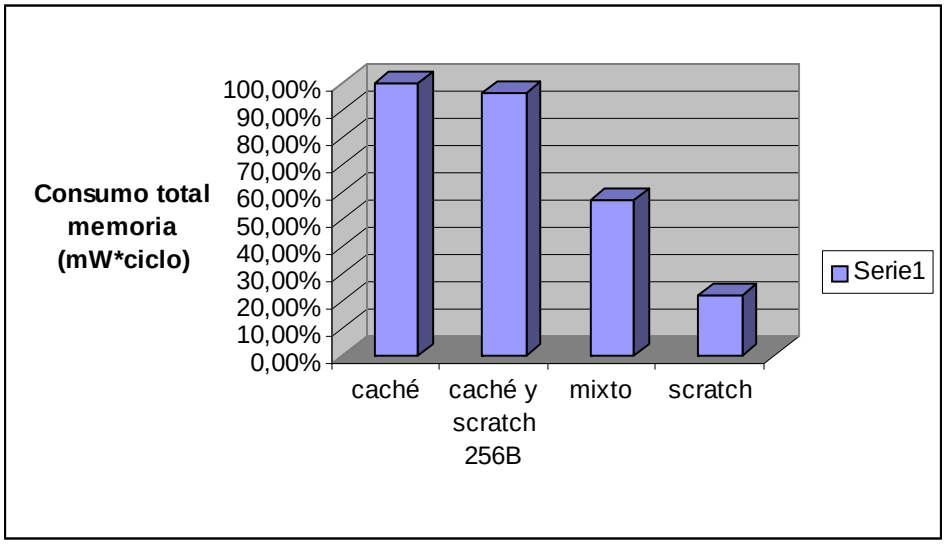

Figura 31: consumo de energía de la memoria (mW*ciclos) vs configuración de memoria

5.3 DCT

El primer benchmark utilizado en el proyecto parte de la base del algoritmo DCT.

La Transformada de coseno discreta (DCT del inglés *Discrete Cosine Transform*) es una transformada basada en la Transformada de Fourier discreta, pero utilizando únicamente números reales.

Algunas de sus aplicaciones, como se explicó en la presentación de los algoritmos, pueden ser JPEG o MPEG.

Algoritmo principal:

```
for (vblk = 0; vblk < blkrow; vblk++) {//movimiento vertical en la matriz
  for(hblk = 0; hblk < blkcol; hblk++) {//movimiento horizontal en la matriz
     inicializar la matriz block de tamaño BxB desde la entrada
     block_dct(block, cos1, cos2)
     asignar la salida desde block
  }
}
```

Algoritmo 10: código original del programa principal del DCT

Función block_dct:

```
for (i = 0; i < B; i++) {
  for (j = 0; j < B; j++) {
     sum = 0.0;
     for (k = 0; k < B; k++) {
        sum += block[i*B+k] * cos2[k][j];
     }
     tmpblk[i][j] = sum;
  }
}

for (i = 0; i < B; i++) {
  for (j = 0; j < B; j++) {
     sum = 0.0;
     for (k = 0; k < B; k++) {
        sum += cos1[i][k] * tmpblk[k][j] ;
     }
     block[i*B+j] = sum;
  }
}
```

Algoritmo 11: código original de la función block_dct

Siendo B=8 en los casos estudiados.

Como se puede ver el algoritmo consiste en ir realizando la función DCT a todos los bloques de tamaño BxB tomados desde la matriz de entrada y asignar desde ellos el resultado a la matriz de salida.

En el algoritmo principal cabe destacarse que se va accediendo a la matriz de entrada bloque a bloque lo que ya puede predisponer a una técnica de prebúsqueda por bloques.

En la función DCT se comprueba que cada uno de los dos bucles externos que la componen se comportan exactamente igual que el algoritmo de multiplicación de matrices. Por tanto cualquier análisis previo que se hiciera para la multiplicación de matrices es válido para la función DCT y de hecho cabría esperar los mismos resultados.

Como se puede ver, en el primer bucle el acceso a los arrays es del orden cúbico para block y cos2, así como a la variable sum, y cuadrado para tmpblk. El patrón de acceso para la block y tmpblk es por filas, mientras que el de cos2 es por columnas. Al dato sum se accede en todas las iteraciones del bucle más interno, es decir es un dato al que se accede continuamente.

Para el segundo bucle se puede decir lo mismo: orden cúbico en el acceso a cos1 y tmpblk y cuadrático en el acceso a block. A block y a cos1 se accede por filas mientras que a tmpblk se accede por columnas. Al dato sum se accede continuamente.

Para este benchmark sólo se hará un tipo de pruebas: con tamaño de los datos igual a 320x240 (NROW,NCOL).

Prueba 1: Con tamaño de los datos (NROWxNCOL) 320x240

Para este benchmark sólo se va a hacer una prueba debido a la configuración del algoritmo. Al realizarse por bloques, realmente el tamaño de la entrada y la salida pierden importancia relativamente dentro de cada iteración interna del algoritmo principal, si bien evidentemente cuantos más datos haya que procesar más iteraciones habrá que realizar. Los datos que se procesan el la función block_dct son matrices de tamaño BxB

Versión 1.

Para esta versión se ha utilizado una memoria caché de 4KB, sin memoria scratch.

El código no se modifica al ser la versión básica que no incorpora scratch, el algoritmo es el original.

Versión 2.

Para esta versión se ha utilizado una memoria scratch de 2KB con una caché mediana de 2KB.

En cuanto al código, se modifica ligeramente de acuerdo al siguiente estudio: Al ser B ocho, y las matrices cos1, cos2, tmpblk y block de tamaño BxB, cada una de ellas tiene sesenta y cuatro elementos, de tipo flotante. Cada uno de ellos ocupa ocho bytes por lo que cada matriz ocupa quinientos doce bytes.

En principio podría parecer que entrarían las cuatro en la scratch, pero al contrario que las demás, y desde el punto de vista de la función block_dct, la matriz block es una matriz de entrada cambiante en cada llamada a la misma. Por tanto, se debería transferir a la scratch antes de cada

invocación. Si se implementase esa solución sin prebúsqueda el rendimiento se degradaría. Y si bien las cuatro matrices entrarían en la scratch, la llenarían completamente y no dejarían espacio libre para realizar la prebúsqueda. Por tanto en la scratch tan sólo se posicionan los datos cos1, cos2 y tmpblk, y la opción de prebúsqueda se reserva para una posible solución posterior.

Por supuesto no se hará prueba sin prebúsqueda aumentando innecesariamente el tamaño de la scratch para no obtener nada a cambio.

Versión 3

Para esta versión se ha utilizado una memoria scratch de 4KB con una pequeña caché asociada de 256B.

Se utiliza prebúsqueda para el dato block. El código se modifica ligeramente para realizarla, el resto de los datos se mantienen fijos en la scratch durante toda la ejecución como en la versión anterior. La prebúsqueda no se realiza dentro de la función blk_dct sino en el código exterior que la invoca. El aumento de tamaño de la scratch es el que permite realizar la misma prebúsqueda que antes no se podría hacer.

De nuevo, para explicar la prebúsqueda no se pondrá el código sino un esquema en pseudocódigo:

```
transferir el primer dato para block desde la entrada

for (vblk = 0; vblk < blkrow; vblk++) {//movimiento vertical en la matriz
  for(hblk = 0; hblk < blkcol; hblk++) {//movimiento horizontal en la matriz
    esperar a que termine la última transferencia cuadrada para block
    prebuscar el siguiente valor de block
    block_dct(block, cos1, cos2)
    asignar la salida desde block
  }
}
```

Algoritmo 12: prebúsqueda de block

Tablas de resultados.

Caché 4KB	Valores
Tiempo de ejecución (ciclos):	6.419.124
Consumo de memoria scratch (mW*ciclos)	0
Consumo de memoria cache (mW*ciclos)	5.937.284
Consumo total de memoria (mW*ciclos)	5.938.396
Numero de accesos a cache:	1.959.263
Numero de accesos a scratch:	0
Numero de accesos a memoria principal:	2.223
Aciertos de cache:	1.957.040

Caché de 2KB y Scratch de 2KB	Valores
Tiempo de ejecución (ciclos):	8.260.179
Consumo de memoria scratch (mW*ciclos)	60.503
Consumo de memoria cache (mW*ciclos)	3.377.627
Consumo total de memoria (mW*ciclos)	3.439.933
Numero de accesos a cache:	1.836.671
Numero de accesos a scratch:	192.000
Numero de accesos a memoria principal:	3.606
Aciertos de cache:	1.833.063

Scratch de 4KB. Prebúsqueda para block	Valores
Tiempo de ejecución (ciclos):	6.470.424
Consumo de memoria scratch (mW*ciclos)	143.869
Consumo de memoria cache (mW*ciclos)	1.101.032
Consumo total de memoria (mW*ciclos)	1.246.812
Numero de accesos a cache:	1.659.777
Numero de accesos a scratch:	275.584
Numero de accesos a memoria principal:	3.821
Aciertos de cache:	1.655.954

Análisis de resultados.

Se demuestra mejor, no en cuanto a la energía pero sí en cuanto al tiempo de ejecución, la versión que sólo incluye caché que la que la mezcla con la scratch. Sin duda el dato no posicionado en la scratch no funciona bien en una caché menor ya que el número de accesos a memoria principal se ve casi duplicado. Este resultado es bastante decepcionante ya que este tipo de configuración es la que ha estado obteniendo mejores resultados durante todas las pruebas, al capturar las ventajas de la scratch sin tener que pagar demasiado las limitaciones impuestas por nuestra implementación.

La versión que sólo tiene scratch e incluye prebúsqueda para el dato conflictivo iguala en tiempo de ejecución a la versión con caché a la vez que disminuye visiblemente el consumo.

Este resultado de igualación del tiempo de ejecución es especialmente interesante, ya que a pesar del comentado mal comportamiento de escalares, índices y otros elementos conflictivos, con los consiguientes accesos de más a memoria principal, el rendimiento de la scratch hace que no haya pérdidas, lo que se puede considerar como un logro

De nuevo el análisis de la energía ofrece mejores resultados para versiones con cachés menores y más accesos a scratch (ver [Figura 33]), aunque desvirtuando los resultados para aquellas que se configuran con una scratch de 4KB.

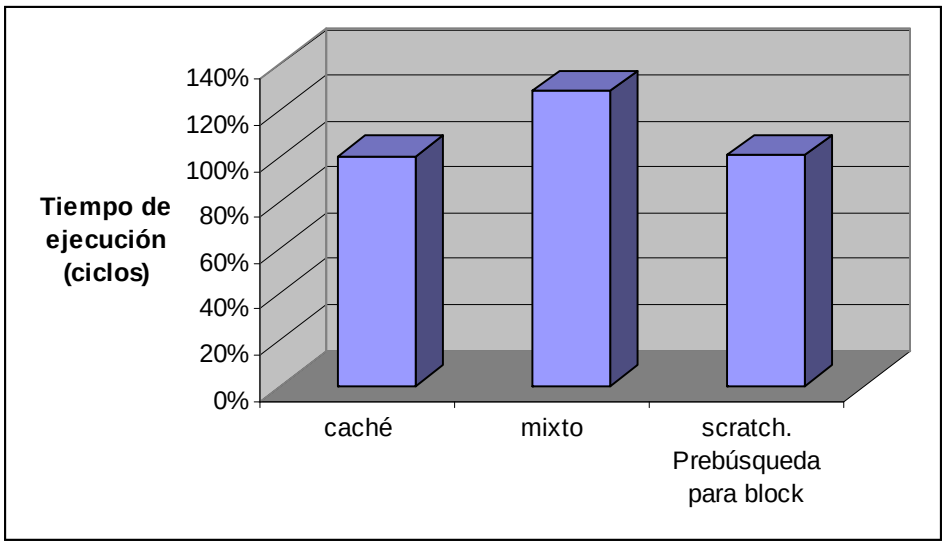

Figura 32: tiempo de ejecución (ciclos) vs configuración de memoria

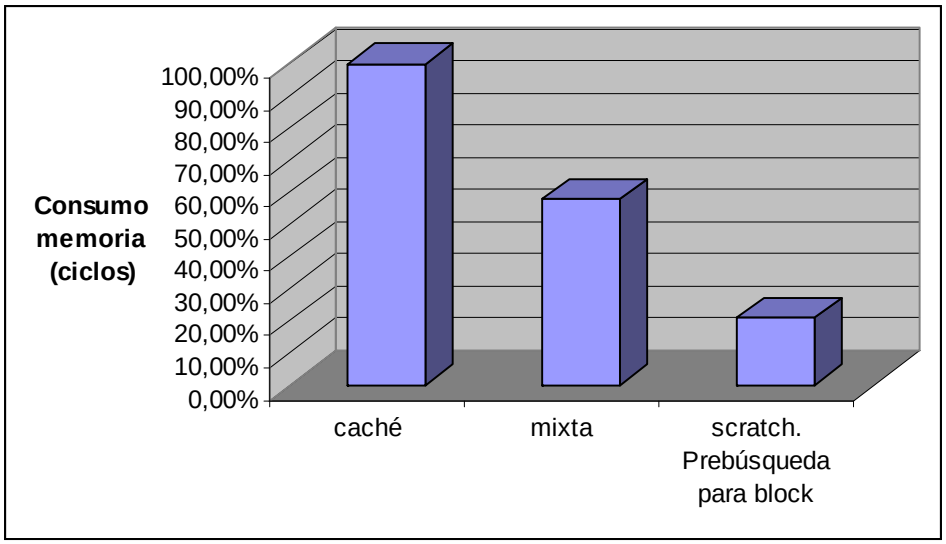

Figura 33: consumo de energía de la memoria (mW*ciclos) vs configuración de memoria

6. VALORACIÓN DEL PROYECTO

Este proyecto surgió con bastantes problemas iniciales. En primer lugar no era el tipo proyecto deseado por nadie del grupo, cuyos componentes siempre se han sentido más afines a las asignaturas del departamento de sistemas informáticos que a las de DACYA.

Ninguno tenía conocimientos avanzados del sistema operativo LINUX y uno ni siquiera había cursado la asignatura de laboratorio de sistemas operativos. Además, al ser asignado en la fase de ampliación de los proyectos no propuestos por alumnos se comenzó muy tarde.

La dejadez propia de los primeros meses hizo que en enero no se hubiese hecho prácticamente nada aparte de leer documentación. Nadie del grupo conocía el simulador. Así, se partió de unos meses desaprovechados casi totalmente. Sin embargo, cuando se empezó a ver más claro el desarrollo y objetivo global del proyecto casi nunca se llegó a una situación en la que no se supiese qué hacer.

Creemos que un grupo de personas más habituado a trabajar en estos entornos, trabajando a tiempo completo, no hubiesen tenido ningún problema en documentarse y acabar la parte de programación añadida al simulador en unas tres semanas. Lo mismo se puede decir de la parte de los benchmarks: cuando se llegaba a un punto en el que los resultados no cuadraban en muchos casos no se supo dar la respuesta correcta por el desconocimiento del funcionamiento interno del lenguaje C. Los benchmarks escogidos son muy comunes y alguien acostumbrado a la realización de pruebas tampoco debería tener problemas con ellos, en dos semanas podrían estar listos. El trabajo de documentación final, a tiempo completo entre tres personas no debería llevar más de una semana.

Por tanto creemos que este proyecto, desarrollado cuidadosamente por tres personas con cierta experiencia inicial, incluyendo el trabajo de documentarse, programar el simulador, realizar las pruebas y crear la memoria y la presentación final podría realizarse en dos meses. No obstante, también pensamos que esas personas que tuviesen conocimientos previos del simulador, de programar en C, de conocer lo suficientemente bien su funcionamiento interno como para saber analizar resultados inesperados, de saber documentarse por sí mismas en poco tiempo, etc, deberían cobrar bien su trabajo. Por tanto hacemos una estimación de un sueldo mensual de dos mil quinientos euros para cada uno de ellos, por lo que siendo dos meses de trabajo, se les debería pagar cinco mil euros a cada uno de ellos.

Por descontado no estamos diciendo que el trabajo por nosotros realizado valga esa cantidad, más que nada porque sin la orientación del profesor director del proyecto no hubiésemos sido capaces de avanzar por nosotros mismos; estamos evaluando el precio que valdría este proyecto en el mundo real.

7. PALABRAS CLAVE

- *RISC*
- *MIPS*
- *RAM*
- *ASIC*
- *DSP*
- *SRAM*
- *SDRAM*
- *PDA*
- *DVD*
- *MPEG*

8. BIBLIOGRAFIA

[1] Stefan Steinke, Christoph Zobiegala, Lars Wehmeyer, Peter Marwedel. "**Moving Program Objects to Scratch-Pad Memory for Energy Reduction**" (2001) Universidad de Dortmund.

[2] Rajeshwari Banakar,Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, Peter Marwedel. "**Comparaison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption**" (2001) Universidad de Dortmund.

[3] J L Hennessy and D A Patterson. "**Computer architecture – A Quantitative Approach**" (1994)

[4] Binu Mathew, Al Davis. "**An Energy efficient high performance Scratch-Pad Memory System**". Universidad de Utah.

[5] Sumesh Udayakumaran, Rajeev Barua. "**Compiler-Decided dynamic memory allocation for scratch-pad based embedded systems**" Universidad de Maryland.

[6] Ozcan Ozturk , Mahmut Kandemir, Ibrahim Kolcu. "**Shared scratch-pad memory space management**" Universidad de Pennsylvania y Universidad de Manchester.

[7] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, Peter Marwedel. "**Assigning Program and Data Objects to scratch-pad for energy reduction"**. Universidad de Dortmund.

[8] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, Peter Marwedel. "**Scratch-Pad memory: A design alternative for cache on-chip memory in embedded systems"**. Universidad de Dortmund

[9] Raul Jimenez. "**Sistemas Empotrados**". Universidad de Huelva.

[11] Todd Austin, Eric Larson, Dan Ernst. "**SimpleScalar: An Infrastructure for Computer System Modeling**". University of Michigan.

[12] **An integrated hardware/software approach for runtime scratchpad management.** www.simplesacalar.com